ARL-MR-0916 ● JAN 2016

US Army Research Laboratory

Cyber Fighter Associate

by Charles Huber and Lisa M Marvel

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-MR-0916 ● JAN 2016

US Army Research Laboratory

Cyber Fighter Associate

by Charles Huber and Lisa M Marvel
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
January 2016	Final	1 June 2014–31 January 2015

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Cyber Fighter Associate	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
Charles Huber and Lisa M Marvel	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
US Army Research Laboratory ATTN: RDRL-CIN-D Aberdeen Proving Ground, MD 21005-5067	ARL-MR-0916
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Cyber Fighter Associate (CyFiA) is a decision-support program that works with the cyber-network modeling and simulation system, CyAMS. The CyFiA proposes agility maneuvers to a user based on cost and utility of the maneuver as well as the network and node facts. The control flow of the program models a human Observe, Orient, Decide, Act (OODA) loop. The OODA loop has been used as a model to describe decision-making in military environments, making it a reasonable model to use in the CyFiA. The CyFiA was tested to accomplish a patch-management mission while securing a critical path. As a first proof of concept a simulation with a network of 10 nodes and 4 possible agility maneuvers was used. In the future more agility maneuvers will be added and network complexity increased. It is projected that a tool like CyFiA will be a decision aid for network analysts and cyber teams evaluating and applying various agility maneuvers while accomplishing cyber missions in a fast-changing, dynamic environment.

15. SUBJECT TERMS

Cyber Fighter Associate, CyFiA, Cyber Army Modeling and Simulation, CyAMS, OODA loop, decision-support program

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Lisa M Marvel
Unclassified	Unclassified	Unclassified	UU	86	19b. TELEPHONE NUMBER (Include area code) 410-278-6508

Contents

List of Figures

Preface

This work was developed during a summer internship between coauthor Charles Huber's junior and senior year in Computer Science at Pennsylvania State University. Huber was approached by Patrick McDaniel during his junior year to become a research assistant with the Systems and Internet Infrastructure Security (SIIS) laboratory at Pennsylvania State. Part of SIIS's ongoing research efforts includes collaboration with the Cyber-Security (CSec) Collaborative Research Alliance (CRA). The US Army Research Laboratory sponsors the CSec CRA, and Huber was given an opportunity to conduct research over the summer with the Computational and Information Sciences Directorate. Thus, this research is part of a larger ongoing body of research directed toward the goals of the CSec CRA.

INTENTIONALLY LEFT BLANK.

1. Introduction

As computer technology becomes more integrated into a Soldier's equipment, it becomes vital to protect software and networks from exploits and attacks. It becomes especially important to secure software and networks during a cyber operation to maintain the confidentiality, integrity, and secrecy of data. Massive amounts of research have focused on detecting, analyzing, and reacting to attacks of all kinds. However, current-day agility is mostly reactive. A network administrator usually must react to an attack, making it very difficult to keep information from being stolen or even keep the network up and running at full capability. A decision-support tool can help an administrator make effective, agile decisions, securing both software and network assets; moreover, this tool can provide a more proactive defense during cyber operations. One such decision tool is the Cyber Fighter Associate (CyFiA).

2. Background

The CyFiA is a decision-support program currently being developed and is based on work done on the Pilot Associate[1] and Warfighter Associate programs.[2,3] The CyFiA proposes agility maneuvers to a user based on cost and utility of the maneuver as well as the network and node facts. The program responds to certain data stimuli to propose agility maneuvers to a user. The control flow of the program models a human Observe, Orient, Decide, Act (OODA) loop. The OODA loop has been used as a model to describe decision-making in military environments, making it a reasonable model to use in the CyFiA.

To test the CyFiA, a simulation was designed. We are using a program called Cyber Army Modeling and Simulation (CyAMS). CyAMS can model on very large-scale networks with the help of a high-performance computing system. CyAMS has demonstrated the ability to model networks containing up to 35 million nodes. As a test environment a mock network of 10 nodes was created in CyAMS; a cost program was written to calculate critical path, patch routing, and agility maneuver cost[4]; a graphical user interface (GUI) was created to display the mock network; and a communication program was written so that the CyFiA and the cost program could communicate with CyAMS and vice versa.[5] Figure 1 accurately displays the communication setup between all 5 parts of the simulation. Details of the protocol design can be found in a 2002 reference work.[6] (The Appendix provides a listing of code.)

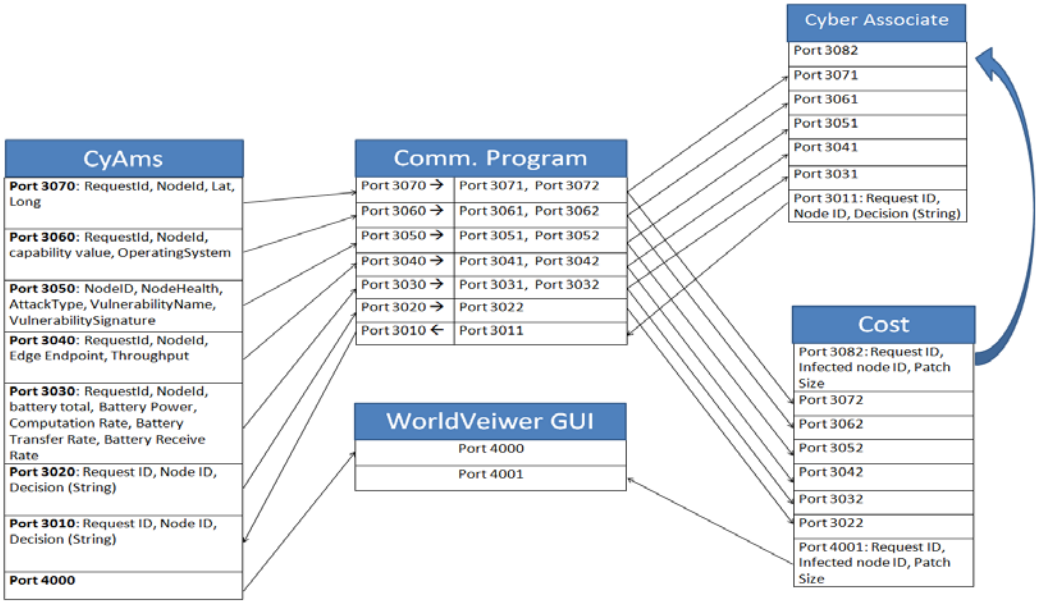

Fig. 1 Simulation communication-protocol layout

3. Cyber Fighter Associate Design

The CyFiA is written using the Java programming language in coordination with an Xbase domain-specific language designed by Veloxiti, Inc., of Alpharetta, Georgia. This combination is known as a Solomon project. Solomon projects run using the Solomon engine. This engine was created by Veloxiti. The Solomon engine runs 2 graphs. The first is the Observe–Orient (O–O) graph; the second is the Decide–Act (D–A) graph. Both graphs have a specific file type: .kb.

The O–O and D–A graphs consist of beliefs, plans, goals, links, and monitors. Beliefs only exist in the O–O graph. Plans and goals only exist in the D–A graph. Links are used to connect beliefs to beliefs or to connect plans to goals and vice versa. Monitors are used as bridges from the O–O graph to the D–A graph. Figures 2 and 3 show small portions of the O–O and D–A graphs, respectively. (Due to their sizes, the graphs could not be shown in their entirety.)

There are other output entities that exist on monitors and plans. Whenever a monitor is triggered, it is possible to notify the user using a notification and by specifying the notification type in a handler. This tool is very useful for both debugging and for understanding how the engine works. Notifications can only be sent from monitors. In a similar way, actions and proposals exist on plans. An action is a form of output where the engine tells other parts of the program to do something. This process was used in the CyFiA when cost values were needed from the cost program. Proposals are similar to notifications, but they serve a

different purpose. Proposals are actions that the engine suggests to the user. Whether these actions are carried out is completely up to the person receiving the proposal. In the simulation, proposals came in the form of agility maneuvers, so multiple proposals were often given. In addition, due to the nature of the simulation, proposals were not used frequently—if at all—because all of the actions were automated and there was no human interaction.

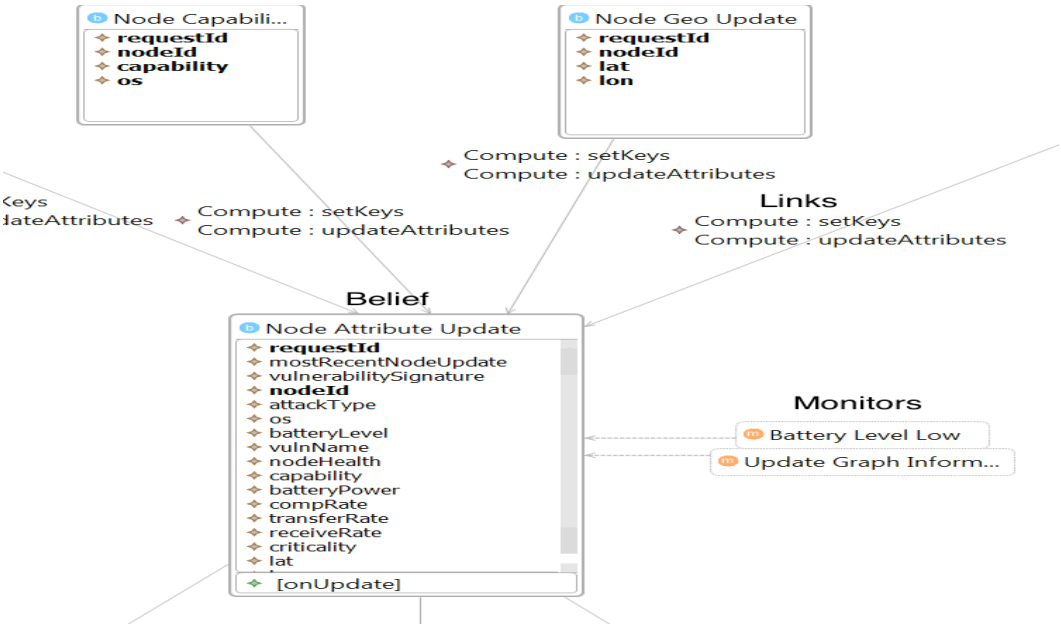

Fig. 2 Beliefs, links, and monitors

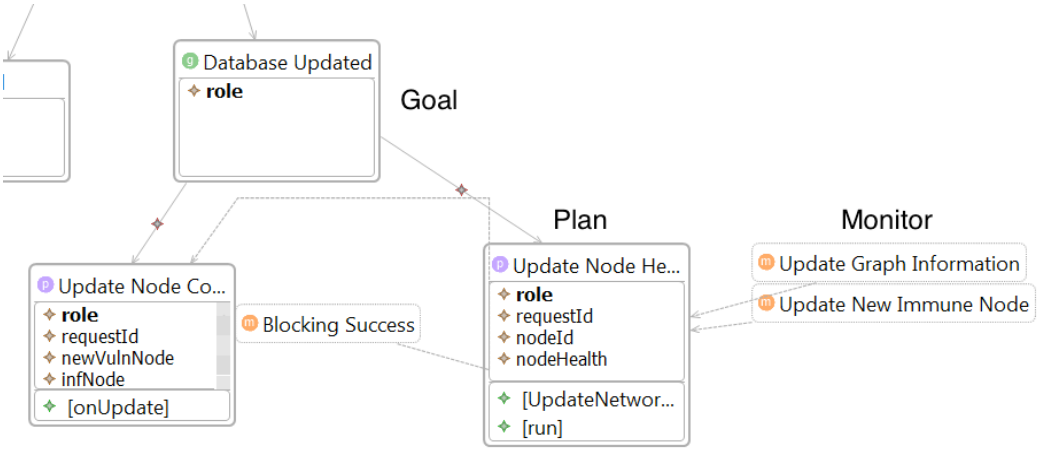

Fig. 3 Plans, goals, and monitors

The Java files were created to run the engine as well as listen for new information. Threading allows the engine to run on its own thread concurrently with other parts of the associate.

4. Interacting with Network Simulations and Cost Programs

A network is made up of nodes that all have varying characteristics. For this reason, the Node class was created. Nodes have the following characteristics: a node identification (ID), the operating system, a capability metric, the node's geographical location, a computation rate, battery power, transfer and receive rates, node criticality, a patch-origin flag, a list of connections, and node health. The node ID was an identifier consisting of a number stored as a string. This allows ID to be simple and the process made passing it through the graphs easy. The list of connections is a list of nodes to which a given node is connected. Node criticality is a Boolean that is true if the node is along the critical path. At this time, node health is a string that represents the state of the node. A node could be infected, vulnerable, susceptible, or immune. For nodes that are infected by the malicious application, the node health is set to "infected". Likewise, nodes that are patched have a health set to "immune". Vulnerable health means that a node has not been infected but is also not immune while susceptible health means the node is vulnerable and also communicates with a known infected (or previously infected node).

Another important part of the CyFiA is the database class. To make fast computations, it is important to have some information ready to use. The database contains a list of all the nodes. Every time a node is created, it is stored in the database along with all of the node's attributes. The database also contains the list of nodes that make up the critical path. This list makes accessing the nodes on the critical path easy and quicker than repeatedly checking a node's criticality. The database is a static entity. This term means that there is only one database. A "getter method" is made to get the instance of the database. Having a static database is important because the database is not bound to any specific class and can be accessed by multiple threads. The database class also has many important methods to keep it and the nodes stored in it updated. The database has the ability to change features in nodes.

Once started, the CyFiA continuously receives input and is constantly processing data. Input is parsed and converted into a belief, but the engine must run from a single thread. To overcome this restriction, a blocking queue was created inside the Request Handler class. This class was initially designed to handle requests that had to be sent to the cost program, and it would also send actions back to

CyAMS; however, this initial design was changed. The Request Handler now sends and receives requests to the cost program and also has a second blocking queue that holds beliefs to be pushed to the engine. The "makeBelief" method is used to take the top belief off the queue and push it to the engine for processing. The Request Handler class, like the database, is also static. The handler needed to be designed so that there would only be one queue for beliefs and only one queue for requests.

CyAMS sends information to the CyFiA via multiple ports. Ports 3031, 3041, 3051, 3061, and 3071 were all used in receiving information from CyAMS. To handle all of the information, the Socket Listener class was made. Information is passed between programs using the User Datagram Protocol (UDP). Because UDP was implemented, setting up port communications was relatively easy—but UDP does not guarantee that the information received is in the order that it is sent. Data were sent as a string of comma-separated values. The function used to receive data from CyAMS is implemented using a series of "if" statements to determine how to parse the information received on each port. Once information is received, it is parsed using the method "setInData". This method parses data as a single string. The string is then split using the split function using a delimiter of a comma and stored in a string array called tokens. Every packet received was expected to contain specific information based on the port number on which it was received. This information was also expected to be in a specific order, so a string array was an easy solution for indexing that data and using it to create beliefs. The Socket Listener creates beliefs and adds them to the belief queue in the request handler. Therefore, the Socket Listener class calls an instance of the Request Handler so that when it receives data from CyAMS, it will create the appropriate belief and add the belief to the queue. The Socket Listener class implements the Runnable Java interface. This means that it can run on a separate thread. Five separate, concurrent Socket Listener threads receive information from CyAMS. A sixth thread is initially run to receive the initial critical path from the cost program, but it is then stopped so that the port can be used again later.

Threading is a large part of the CyFiA; to manage all of the threads, the CyFiAJob class was made. This class has 3 purposes. The first purpose is to instantiate and run all of the socket listeners. The second purpose is to constantly check the Request Handler class's belief queue. If the queue has a belief in it, the class makes a call to the Request Handler's "makeBelief" method. This method takes the top belief off the queue, determines which belief to create, creates the belief, and updates the engine. Creating beliefs inside the CyFiAJob class ensures that the engine is constantly being updated by the same thread even though the Request Handler queue can be updated in many threads. The final purpose of

CyFiAJob is to create a thread for sending an action back to CyAMS. This thread runs the AutomatedActionSender class. In addition to the socket listeners and the CyFiAJob class, there are 3 other classes that all run inside a single thread provided by the Solomon engine. The engine provides 2 different types of output providers. The first is a basic output provider. This type of output provider blocks the engine every time an action, notification, or proposal has to be created and output. The second type of output provider is the threaded output provider. This provider creates separate threads for notifications, proposals, and actions. In the CyFiA, the threaded output provider is used.

The Notification Handler class handles all of the notifications inside the threaded output provider. The handler also holds the custom code that makes a notification. This design is useful because notifications can take different forms; therefore, writing custom code for each different notification is necessary. When a notification is instantiated on a monitor, it is added to a queue. Using a special method called “process” that is provided by the output provider, a notification is pulled off the queue. The notification handler determines what kind of instance the notification is by using a series of if-else statements. Once the correct notification instance is found, the custom notification code is executed and a notification is displayed. This process works the same way for proposals, as well.

The Proposal Handler class handles all the proposals inside the threaded output provider. The Proposal Handler functions exactly like the Notification Handler. The CyFiA has a simple proposal handler and does not output proposals because the simulation did not need proposals to be displayed. The CyFiA will eventually have many proposals for each agility maneuver.

The Actions Handler class is the most complex of the handler classes. When an action is created and sent from a plan, it is put in a queue. Using the method process (just like notifications and proposals), the top action is taken from the queue and is checked against a series of if-else statements until the correct instance is found. The action is then executed. Actions are complex because some require computations and may even make beliefs. For this reason, the Action Handler class has an instance of both the Request Handler class and the database. There are 7 actions handled by the handler. The first action is named “CheckCriticalNodes”. This action is called when a new infected node is introduced to the engine. Once a new infected node is detected (received from CyAMS), the critical path is checked for nodes whose health is susceptible. The susceptible node health means that the node is directly connected to an infected node. It is important to make sure the critical path is secured, so it is important to perform agility maneuvers on susceptible and vulnerable critical nodes first. Therefore, if a critical node is susceptible, a belief is created to update the engine

and do agility maneuvers on the susceptible node. If a susceptible critical node does not exist, the function will search the critical path for vulnerable critical nodes, and agility maneuvers will be performed on a particular, but arbitrary, vulnerable critical node. If the critical path is secured, the search to find a vulnerable or susceptible critical node should not run. In addition, it is necessary to query the D–A graph to determine whether the action is being executed without a new infected node being passed into the engine. (At the writing of this report, this particular query has a defect that occurs for an unknown reason. It can cause redundancies in engine execution and lead to multiple outputs for a single input and sometimes infinite loops. This complication will be described in more depth in Section 7.)

The second action in the handler is "UpdateNetworkHealth". The UpdateNetworkHealth action changes nodes health throughout the network inside the database. For example, if Node 3 became infected and Nodes 4–6 were all not immune, their node health would be updated to vulnerable or susceptible depending on whether they are connected to the infected node. This action is also used to update a single node's health to immune. If a node is patched or healed, then it becomes immune and needs to be updated in the database so that all information is up to date. The next 4 actions are "IPBlockCost", "WallOffCost" or Quarantine, "BestThroughputCost", and "HealSoftwareCost". All 4 of these actions have the same purpose, and they all send a request to the cost program requesting costs for the specific agility maneuver. The 4 agility maneuvers represented by these actions are Intrusion Prevention (IP) blocking; function quarantining; patching via a best throughput path; and software healing. (These maneuvers will be explained in more detail in Section 5.) Each of these 4 actions blocks and waits for a response from the cost program. After a response is received from the cost program, the action continues by adding an "automated action" to the Automated Action Sender class list. The last action handled by the actions handler determines how many agility maneuvers are possible given any particular node. The maximum number of agility maneuvers for this simulation is 4: IP blocking, function quarantining, software healing, and patching. However, there may be times in which certain agility maneuvers are just not possible. For example, if a patch file does not exist, it would be impossible to patch. In addition, if a newly infected node is being handled, it would be impossible to implement an IP block because a node cannot be blocked from itself. The last action in the Actions Handler is "StartActionSender". "StartActionSender" gets the instance of the Automated Action Sender and calls the function "getNumActions" that runs a series of checks and queries into the graph to determine which agility maneuvers are valid; then, it set a variable to the number of agility maneuvers possible.

The Automated Action Sender class is the decision-making class. This program decides which agility maneuver to execute in the simulation. In its final form, the CyFiA may have a class similar to the Automated Action Sender but it probably will not act on decisions automatically. The Automated Action Sender class also implements the Runnable Java class and contains the run function. There is a "while" loop inside the run function that is always executed. There is an "if" statement inside the "while" statement that checks whether a list called "actionList" has a size equal to the number of actions determined by the function getNumActions, which is called in the actions handler. If getNumActions is never called, then the "if" statement will never be true and no actions will be sent from the action list. The action list is a simple array list that holds "AutomatedActions". This class is a simple object class that creates an object tailored to match the input CyAMS is expecting. An AutomatedAction has 5 characteristics: request ID, a starting node, an ending node, the action to be executed, and the cost. Every time a cost is received by one of the 4 agility cost functions in the actions handler (described earlier), it is added to the Automated Actions Sender's action list. Once the list size equals the number of possible automated actions, a sorting function is run. This simple sorting function compares the costs of each agility maneuver and places the maneuver with the lowest cost at the first index of the list (Index 0); then, the Automated Action Sender opens a socket to CyAMS on Port 3010 and sends the action back to CyAMS. After this process is done—if the send is successful—the Automated Action Sender checks to see which action was sent and creates one of 2 beliefs. The first belief is for IP blocking, and the second belief is for all other agility maneuvers. Two separate beliefs are required for the following reasons. There are 2 distinct types of agility maneuvers: software and network. Software-agility maneuvers directly affect a node's software and applications. In this simulation, all software-agility maneuvers changed a node's health to immune. However, in network-agility maneuvers, a node's health would be changed to vulnerable if it was not so already and if it was not infected. Once the proper belief is created, it is sent to the request handler's belief queue to be updated into the engine.

The main class of the CyFiA is tasked with creating the engine, creating all of the handlers for the threaded output provider, and starting the threaded output provider. The main class also threads and starts the CyFiAJob class. This creates a thread-like hierarchy, although child threads run concurrently even with their parents. Child threads in this context refer to the threads that are created within the "main" function or from other threads. Figure 4 shows the hierarchical view of threads from their instantiation in CyFiA.

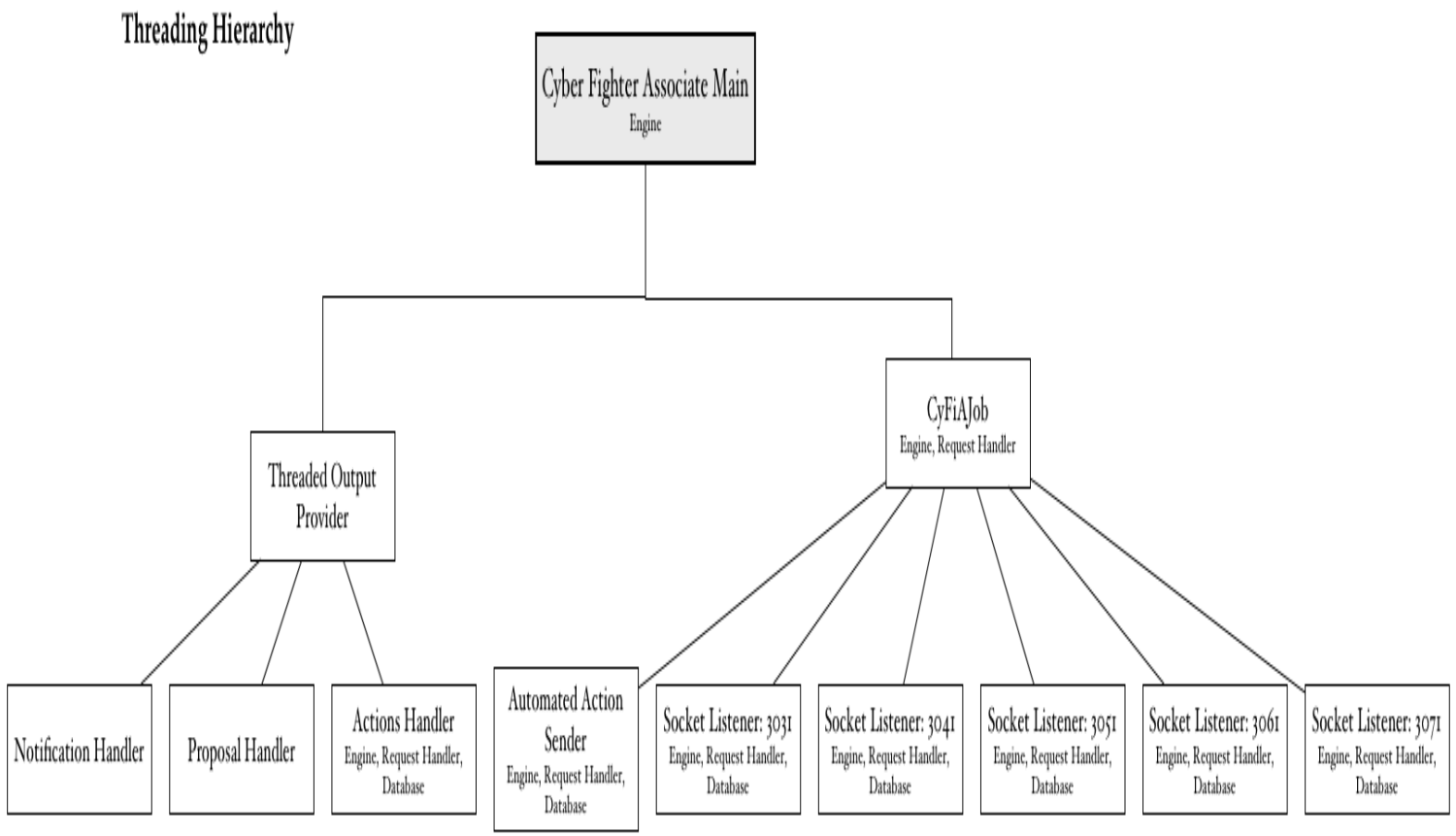

Fig. 4 Threading and class-access distribution per thread

5. O–O and D–A Graph Design and Implementation

The Solomon engine uses advanced data filtering to make decisions. As described earlier, it performs this function by modeling an OODA loop using 2 graphs, the O–O graph and the D–A graph. Beliefs, plans, and goals have 2 special types of values: keys and attributes. A node of the graph will not be created unless all of the keys have a value, but attributes can remain null. In addition, a key for a particular node can be an attribute for another node and vice versa. Using a similar design to the Warfighter Associate, one key is shared by all beliefs, plans, and goals for the purpose of ID. This key is named "requestId". The requestId is particularly useful when querying specific nodes in the graph. Links and monitors are used to exchange data from node to node and from graph to graph, respectively. Links and monitors also have special constraint functions that act as "if" statements to determine whether a link or monitor should be traversed. For example, if a link constraint fails, then the child node that is connected to the link will not be updated. Link constraints work the same way for monitors. The O–O graph takes on a very hierarchical form, whereas the D–A graph has 3 distinct graph sections as shown in Fig. 5. The first section of the D–A graph contains plans and goals to maintain situational awareness (SA). The second part of the D–A graph contains plans and goals to complete the cyber operation. The last division of the D–A graph contains plans and goals to execute agility maneuvers.

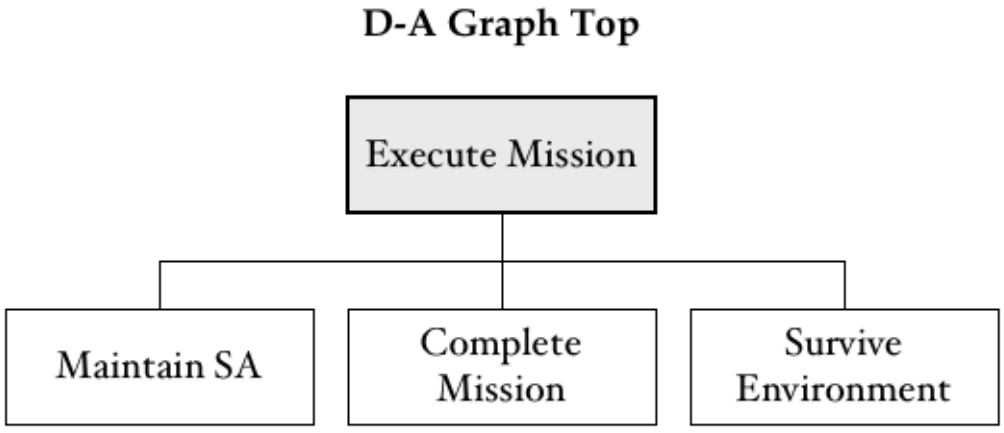

Fig. 5 D–A main goals

The O–O graph has 4 base beliefs: Node Health Change, Node Capability Update, Node Geo Update, and Node Battery Update. These 4 basic beliefs were created to align exactly with the Socket Listener class. Because specific information is

passed on each port, it makes sense to have individual beliefs corresponding to each individual port. For example, if data are received on Port 3051, a Node Health Change belief will be created and added to the request handler's queue. When that belief is taken off the queue, the request handler will see it as an instance of Node Health Change and update the engine with the proper belief. This makes getting information into the O–O graph very easy. All 4 base beliefs link to a single belief called "Node Attribute Update".

Node Attribute Update is the largest belief in terms of attributes because it pools all of the information and is connected to 2 monitors and 3 child beliefs. The most important part of this belief is the monitors. The first monitor is "Battery Level Low". This monitor is only fired if a Node Battery Update was received and the battery level received was below a threshold of 25 (25% of battery capacity). If the battery level drops to or below the threshold of 25 and the node is part of the critical path, then a request would be sent to the cost program for a new critical path because it is possible the node does not have sufficient battery to complete the mission. At this time, this agility maneuver is not implemented but could be in the future.

The second monitor is "Update Graph Information". This monitor is very important because it leads to the SA section of the D–A graph. The SA subtree of the D–A graph is a rather simple but very important part of the CyFiA because it keeps the database updated. The monitor updates the plan "Update Node Healths", which is a child under the goal "Database Updated". The Update Node Healths plan is where the "Update Network Health" action is instantiated. Every time the Update Node Healths plan is updated, the Update Network Health action is executed. Therefore, the only time the plan should be updated is when there is a node whose health has changed to infected, which is a constraint on the monitor Update Graph Information. The monitor will not update the Update Node Healths plan unless the current node being handled by the CyFiA is an infected node. This prevents unnecessary runs of the Update Network Health action and limits actions to run only when the network nodes' health need to be changed. An example of how Update Node Health interacts is shown in Fig. 6.

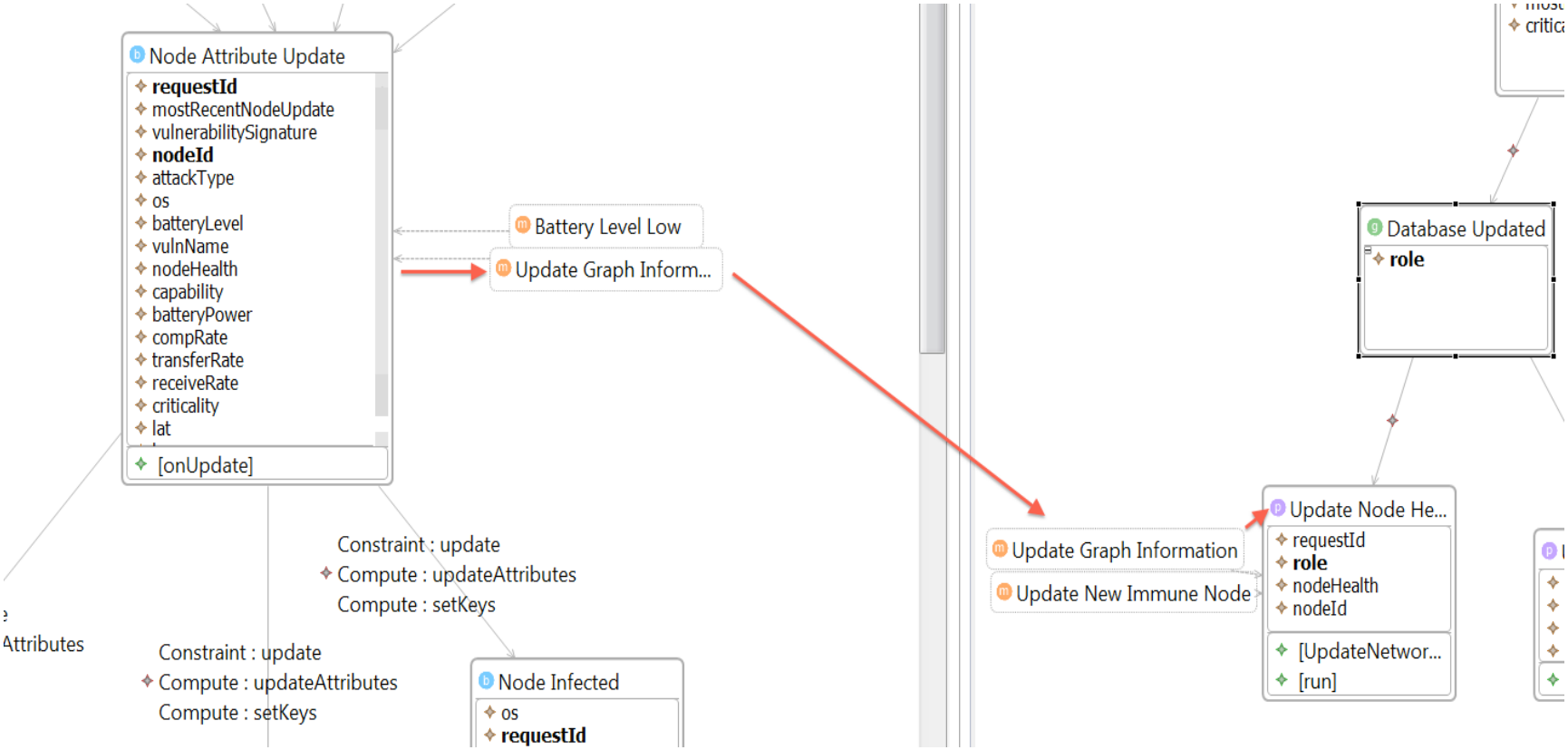

Fig. 6 Belief-to-plan flow

The need for Node Attribute Update as a belief is not really required, and in the future it will be removed for better data-filtering techniques. However, the belief does have 3 child beliefs, all with conditional links and all that are similar in nature. The 3 child beliefs are “Node Vulnerable”, “Node Susceptible”, and “Node Infected”. These beliefs are exactly as they are titled and are traversed only if the node health of the current node being handled by the engine matches the respective belief (so a vulnerable node will cause the Node Vulnerable belief to be updated but none of the others). All 3 beliefs have their own separate monitors that connect to the same plan: “Survive Environment”. However, the monitors are different in terms of constraints. The initial constraint on the “Node Health Change Infected” monitor caused the monitor to fire even if the belief had not been updated. This would cause an infinite loop and a redundancy in agility maneuvers for nodes that were already being handled. A simple query was added to the monitor that would query all Node Attribute Update beliefs with a current request ID. If more than one Node Attribute Update exists with the same request ID, then the monitor will not be traversed. This ends the infinite loop. The other 2 monitors do not have this restriction because the walks through the graph when these monitors are used are not the same as the walk when Node Health Change Infected is fired. In addition, all 3 of the node-health-change monitors cause notifications to be added to the threaded output provider’s notification output queue (as shown in Fig. 7).

```
// Checks to make sure monitor has necessary keys before firing
constraint fire
{
    var boolean okayToUpdate;
    okayToUpdate = false;
    var NodeAttributeUpdate query = new NodeAttributeUpdate();
    query.setRequestId(source.requestId);
    var List<NodeAttributeUpdate> resultList = graph.find(query);

    if ((source.nodeHealth.equalsIgnoreCase("infected")) && (source.nodeId != null) &&
    {
        okayToUpdate = true;
    }
    return okayToUpdate;
}

compute updateAttributes
{
    target.mostRecentVulnerabilitySignature = source.vulnerabilitySignature;
    target.mostRecentVulnerabilityName = source.vulnName;
    target.mostRecentNodeId = source.nodeId;
    target.mostRecentAttackType = source.attackType;
    target.mostRecentOs = source.os;
    target.mostRecentRequestId = source.requestId;
    target.mostRecentNodeHealth = source.nodeHealth;
    target.mostRecentNodeCriticality = source.criticality;
    target.mostRecentCapability = source.capability;
    defaultUpdateAttributes();
}
initialize nodeIsInfected myNotification
{
    // Calling the defaultInitNotification allows us to get the
    // auto-generated assignments.
    defaultInitNotification(myNotification);
    // Then add our customized assignments
    myNotification.notificationDescription = "Node Infected";
    System.out.println("Notification initialized");
```

Fig. 7 Node health change "infected" with notification example

The 3 health beliefs are the last beliefs in the O–O graph that are not singletons. Due to the nature of the simulation, complexity beyond that point could not be added without further research. The simulation's main focus was to show agility maneuvers in the event that any particular node became infected by a malicious application; so, node health was seen as the most important characteristic in a node and also as the only beliefs that would cause agility actions. When any of these 3 beliefs was updated, the monitors attached to these beliefs would connect and update a plan in the D–A graph, Survive Environment.

Survive Environment is the parent to 2 different branches of a tree, the left and the right. The left branch contains agility maneuvers, and the right branch determines on which node the agility maneuvers are performed. In its initial design, the CyFiA would receive an infected node-health update from CyAMS. It would then traverse the graph to perform agility maneuvers on that node. Further discussion into how agility was to be implemented and the overall goal of agility led to a new implementation: Instead of performing agility maneuvers on infected nodes, agility maneuvers would be done to protect critical nodes on a critical path. If a critical node became infected, then an agility maneuver may be done on that node, but not always immediately. Therefore, the right branch of the Survive

Environment tree is always traversed first. This branch leads to a plan called "Secure Critical Path". In this plan, the action "Check Critical Nodes" is created and sent to the action handler for execution. Once Check Critical Nodes executes, a new Node Attribute Update belief is made and added to the Request Handler class's belief queue to be pushed to the engine. The new belief will contain the same request ID, but the node ID and node health will be different. The node will also be critical unless all of the nodes along the critical path are immune to the malicious application. The Check Critical Nodes action will not select another infected node even if a critical node is infected. This is a design flaw in the CyFiA and will be fixed in future iterations. The left branch of the Survive Environment tree is not traversed.

At this point, all of the node health in the database has been updated. Although the monitor Update Graph Information seems as if it could be linked to Node Infected, this could possibly cause a race condition between it and Node Health Change Infected. The database needs to be updated before any agility maneuvers are considered because if the database does not have up-to-date health for all nodes, then selecting a node on which to perform an agility maneuver is limited only to the new, infected network node. Therefore, the Update Graph Information monitor needs to fire and the database needs to be updated before the agility maneuver occurs.

Once the new Node Attribute Update is pushed into the engine, the belief tree is once again traversed, but this time no monitors connected to Node Attribute Update fire because of constraints on those monitors. In addition, either the Node Susceptible or the Node Vulnerable belief will be updated. These beliefs are exactly the same, although keeping them separate helps for better visualization and neatness. Both beliefs also have their respective monitors: "Node Health Change Susceptible" and "Node Health Change Vulnerable". These monitors are similar to the Node Health Change Infected monitor because they also have notifications that are created and output when the monitor is fired. Both monitors also target the Survive Environment plan. When Survive Environment is updated by Node Health Change Susceptible or Node Health Change Vulnerable, the left branch of the Survive Environment tree is traversed and the right branch is not updated. This is due to constraints put on the links between Survive Environment and each branch. The link between the left branch and Survive Environment checks 2 things: the current node and the critical path. The link will only be updated if the current node ID is that of a critical node or if every node in the critical path is immune. The database provides a method that checks every node in the critical path. If any critical node does not have an immune node health, then the method returns false. This means that this link also has the instance of the

database. The idea behind having the constraint on this link is that if the entire critical path is immune, then it should not matter on which node an agility maneuver is performed because the cyber operation will be successful. Therefore, if a new infected node is received from CyAMS, and if the critical path is secured, agility can be done directly on that node immediately.

The left branch begins with a goal: “PERFORMANCE Each Thread Managed”. This goal is inspired by a similar goal in the Warfighter Associate with the same title. Both goals have a similar purpose because they both head off the agility branch of their respective programs. The child plan under the threat-management goal is “Agility Maneuvers”, which has spawned 2 trees. Initial design of the CyFiA had all agility maneuvers grouped under one general goal, but secondary design divided agility maneuvers into 2 classes: network (Fig. 8) and software (Fig. 9). This design allows for more flexibility when determining which agility maneuvers to propose to a user. There may be some scenarios in which software agility is impossible, and the same situation is true for network agility. By separating them into subtrees, only one check is needed to determine whether software or network or both types of agility is needed. This type of design also allows more than one agility maneuver to be selected. A user could decide to do both a network and a software maneuver to secure the critical path or network. The CyFiA has 5 agility maneuvers, 3 software and 2 network, although only 4 have been implemented. In the CyFiA, the agility maneuvers are actions that are executed when specific plans are executed.

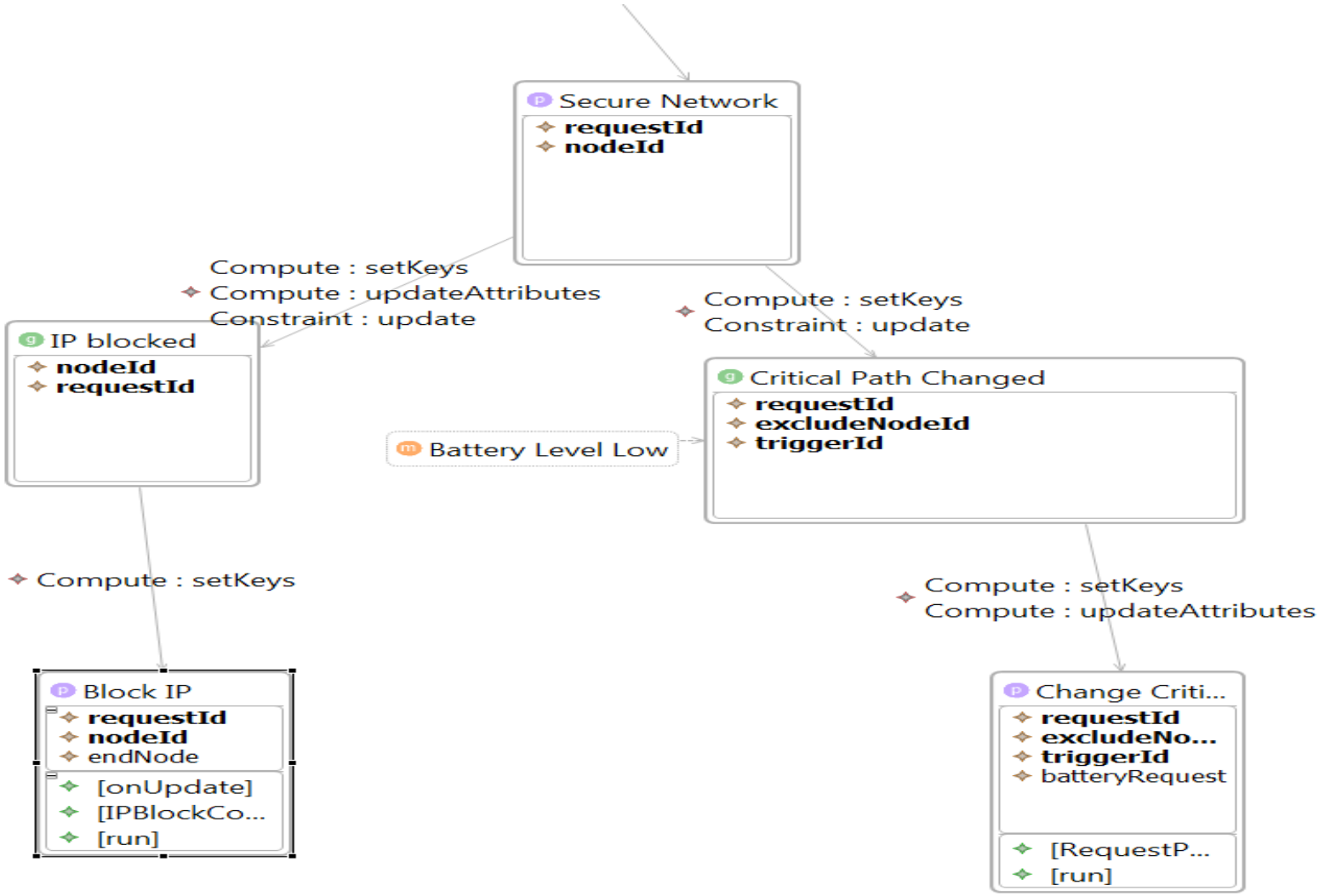

Fig. 8 Network agility maneuvers

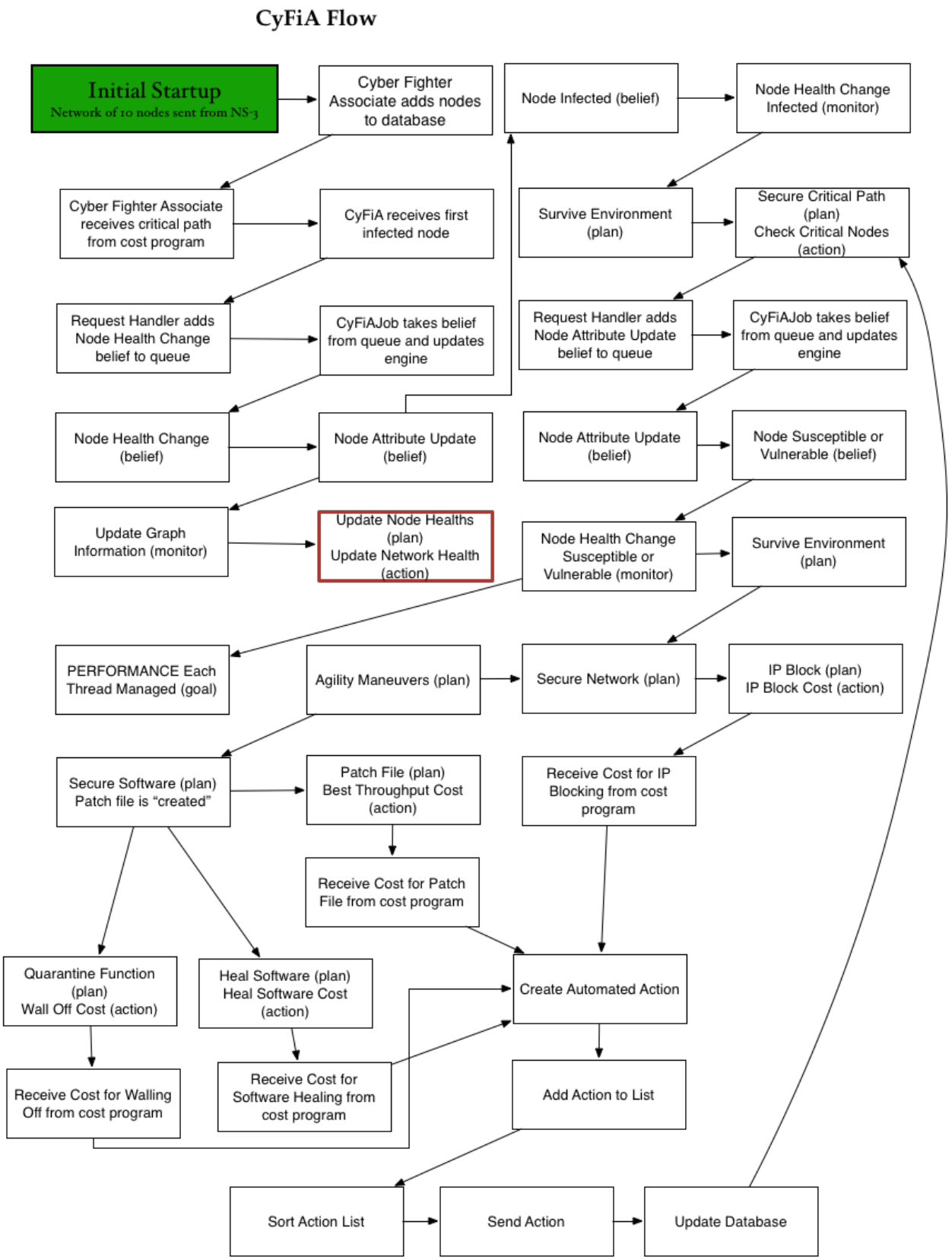

Fig. 9 Cyber Fighter Associate simulation flow

Network agility is agility that directly affects the edges and shape of the network graph. In the CyFiA, this agility set contains the "IP Block" plan and the "Change Critical Path" plan. Network agility does not have the ability to change a node's health to immune. The IP blocking is an agility where the connection between one node and another is completely severed. In the simulation, this was represented by a total throughput of zero. This agility maneuver would change a node's health from susceptible to vulnerable. Although this action does not seem to solve any problems in terms of security, it may cost less in terms of battery and time. The IP blocking is more of a quick-fix maneuver, and it will often be undone after a node has been patched or healed. Once the IP Blockplan is updated, the "IP Block Cost" action is created and executed by the actions handler. The second network-agility maneuver, the Change Critical Path, would send a request to the cost program to request a new critical path, excluding specific nodes. Initial design of the CyFiA would request a critical path when a critical node became infected or when a critical-nodes battery fell below 25% capacity. The critical path could not include infected nodes, and all nodes must have ample battery life. (At the writing of this report, pathing algorithms in the cost program were not complete; therefore, this agility maneuver was left for future work.)

The software-agility walk of the "PERFORMANCE Each Threat Managed" tree is slightly more complex than the network-agility walk. The original design of the CyFiA had a Java class that used a pseudorandom generator to determine whether a patch existed and returned a file name if one did. During the course of this research, the team abandoned this idea and the class was deleted. Instead, a patch file was created on the goal "Software Secured". Using the engine's "onUpdate" method, a patch file name and size are set as attributes inside the Software Secured goal. Both the patch file name and size must be attributes because the goal must exist (have a value for all of its keys) before the onUpdate method is called. Otherwise, the goal would not exist and all child software-agility maneuvers would not exist. After a patch file and size are set, the Software Secured goal links to a plan "Secure Software".

Secure Software branches off into the 3 software-agility maneuvers "Wall-off/Quarantine Function", "Heal Software", and "Patch File". There is also an action that lives on the plan Secure Software, "Start Action Sender". As described earlier, this action calls the "Automated Action Sender" class method getNumActions, which determines how many agility maneuvers the sender is waiting for before it sorts and sends the lowest cost action to CyAMS. This action exists here because of the way the graph is traversed inside the engine. Network-agility maneuvers are always traversed first in a post order fashion. There could already be one agility maneuver in the sender queue by the time the total number

of actions is set, but this is not a problem because the list that contains the maneuvers must be equal to the number of actions for the Automated Action Sender to start the sending process. The number of actions is initially set to –1, so the sending process will not start until the getNumActions method is run at least once. After this, the action-list size will always be equal to or less than the total number of actions. The getNumActions will always be set to at least 2 actions or more. In the simulation, walling off a function and healing software are always possible agility tasks. Therefore, the number of actions could always be expected to be at least 2.

The software-agility plans for walling off/quarantining a function, patch file, and software healing have the same plans as those for the network-agility plans. All 3 functions have an action associated with obtaining a cost from the cost program to associate with the agility maneuver. Once the cost is received by the action, an "Automated Action" is created and added to the sender's queue. Once the action-list size equals the number of actions expected, the list is sorted and the lowest cost action is sent back to CyAMS. After the action is sent to CyAMS, a singleton belief is created, depending on whether the maneuver selected was network or software, and then the database is updated based on this belief.

6. Cyber Fighter Associate Simulation

Our generic simulation begins when CyAMS creates a network of 10 generic nodes. For simplicity, all 10 nodes are identical. Facts about the 10 nodes are sent to the cost program and to the CyFiA. The CyFiA adds all of the nodes and their attributes to the database. This setup is complicated by the fact that the information from CyAMS comes in the form of "request ID, node ID, edge endpoint, throughput". The edge endpoint is the node ID of a node that is connected to the node ID. For example, if {0,1,2,10} was received, then Node 1 would be connected to Node 2 with a throughput value of 10. This means that on any given packet, 2 nodes are received. The CyFiA will add a node to the database if it does not already exist in the database. In addition, the CyFiA will add a connection to a node's connection list if that node is not already connected to the edge endpoint. After all 10 nodes are added, a start node and an end node for a critical path are selected in CyAMS. The start and end nodes are sent to the cost program to determine the best critical path. Once the critical path is calculated by the cost program, it is sent to CyAMS and to the CyFiA. This changes the criticality of the nodes that are along the critical path inside the database. Once this is complete, the associate is ready to evaluate agility maneuvers.

For the patching mission where an infected/vulnerable node is discovered in the network, the state of a randomly selected node is set to infected/vulnerable. A randomly selected node contains the patch for that infection/vulnerability and the state of that node is set to immune. This information can be used by the pathing algorithms to determine the best way to patch the infected/vulnerable node.

There are many different ways a patch file could be distributed in the network. A patch file could be routed to a node using nodes that have the best throughput, but it could also be routed using nodes that have good battery budget. The pathing algorithms had not yet implemented this feature into the cost program, so it was decided the path would be an arbitrary attribute and kept in the CyFiA when that option is added.

After the initial immune node is selected, an initial infected node is selected. The infected node is infected by a generic malicious application. Once this node is selected, CyAMS sends data to the CyFiA, which receives this information on Port 3051. The information comes in the form {request ID, node ID, node health, attack type, vulnerability signature}. The attack type and vulnerability signature are attributes used to provide legible output to a user. They are unimportant in terms of the simulation, but they are important in terms of the agility. The node health of the node ID is infected. This information is received by the CyFiA inside the Socket Listener class. The listener creates a "Node Health Change" belief for the new infected node and adds it to the request handler's belief queue. The CyFiAJob class checks the belief queue, takes the belief off the top of the queue, and updates the engine. The engine updates the proper belief, Node Health Change. Then, the Node Health Change updates the Node Attribute Update belief. The Update Graph Information is then fired, and the Update Graph Information updates the plan Update Node Healths. Then, the Update Network Health action is added to the threaded output provider's action queue and executed. All of the nodes in the network now have updated health based on their connections in the network.

In the O–O graph, the Node Attribute Update links to the Node Infected belief. The Node Health Change Infected monitor is then traversed, updating the D–A plan Survive Environment. The constraint that connects Survive Environment to PERFORMACE Each Thread Managed fails, but the constraint that connects Survive Environment to Secure Critical Path succeeds and eventually updates the plan. The Check Critical Nodes action is then initialized and sent to the actions handler. The action handler executes the action and creates a new Node Attribute Update belief for a critical node that needs to be secured and adds that belief to the request handler's belief queue. CyFiAJob takes the belief off the top of the queue and updates the engine. The existing Node Attribute Update belief is

updated with the same request ID as before but with a different node health and node ID. Both monitors linked to Node Attribute Update are not fired because the constraints fail. Likewise, the constraint to update the Node Infected belief fails; so, it is not updated. Although this constraint fails, either the constraint leading to the Node Vulnerable belief or the constraint leading to the Node Susceptible belief is satisfied and the link is traversed. A monitor is then fired and the Survive Environment plan is again updated. With this update, the constraint on the link leading to the Secure Critical Path plan fails and the Secure Critical Path subtree is not updated. The constraint on the link leading to the agility maneuvers is successful and agility begins.

The IP Block plan adds an action to be executed by the action handler. The handler executes the action requesting a cost for IP Blocking. At the same time, the engine has started traversing the software-agility maneuver tree. The getNumActions method is called, and the number of expected agility actions is set inside the Automated Action Sender class. The cost program sends back a cost for the IP Block plan, and the Automated Action is added to the action list. At this time, more actions are simultaneously added to the action handler's queue to request the costs for patching, software healing, and function quarantining. After the cost program sends back costs for all 4 agility actions, the Automated Action Sender sorts the list and sends the action at the beginning of the list to CyAMS. After the agility action is sent to CyAMS, a new belief is created depending on the nature of the agility (network or software), and the database is updated with a new node health for the node being handled. The process then repeats until all of the nodes in the network are immune to the malicious application.

7. Considerations and Future Work

7.1 Considerations

The CyFiA's current design as described in this report is a first step in a proof of concept. The segmentation of the programs (CyAMS, Cost, and Communication) presented some issues. For instance, the decision to use UDP as a way to communicate proved to be simple but ineffective. Cost and routing data could be calculated within the CyFiA; at least, for small simulations. This could eliminate any overhead with sending the information to an outside program for calculation. However, the limit on the size of networks' simulation in which CyFiA could reasonably handle the computation is unknown at this time.

Another major issue with the current CyFiA implementation is that it is crafted specifically to the simulation. Many key features that are needed for scalability as well as functionality were excluded. As explained earlier, there is no way to alter the critical path because the action is not yet implemented. Pathing algorithms never reached the capability of altering the critical path, and no action was written to handle such risk. Another downfall of the simulation is that it focused solely on node health. This made characteristics of nodes, such as geographical location, nuances that were poorly handled by the CyFiA—if the characteristics were handled at all. Although the database does have methods to update all of these features, beliefs were not created to successfully filter the data.

Node health is an important issue on its own. CyAMS provided only one kind of attack in the form of an infection/vulnerability. Node health was passed through the O–O and D–A graphs as a string variable. This logically means that if a node was to become immune to the malicious application, then it could never become infected again. However, if there were to be more types of attacks, we would see that the node would be immune to both types of attacks despite applying agility for only one attack. Therefore, future design of the CyFiA will most likely keep pairs of <vulnerability, health> or a vector so as to keep track of node health.

An entire segment of the CyFiA is incomplete. The "Complete Mission" subgraph of the D–A graph was never finished. The Complete Mission subgraph is supposed to determine whether all of the nodes in the network have been patched; however, this was seen as a minor function at the time.

7.2 Future Work

With the massive amount of information that will eventually be used with the CyFiA in mind, a Publish and Subscribe Service (PASS) server would be a better option for supplying input to the associate. The predecessor of the CyFiA, the Warfighter Associate, uses a PASS server to supply new information, and it is believed that this method will work equally as well for the CyFiA.

Future work to the CyFiA will also include improvement to the O–O graph for better data filtering, including the implementation of geographic updates, battery-level updates, and node-capability updates. More beliefs will be needed to provide better filtering. The Node Attribute Update node will most likely be deleted and replaced by many beliefs. The "Complete Mission" part of the D–A graph will also be completed. In addition, reasonable decisions for the CyFiA need to be made for the situations when no new information is available. This section of the D–A graph will eventually address that issue.

The total number of scenarios the CyFiA can handle will be greatly increased. Patching is not the only cyber-agility mission that should be addressed. There are quite a few missions that will be researched and implemented into the CyFiA so that agility maneuvers' costs and utility can be evaluated for situations as they occur.

The CyFiA will also be outfitted with a GUI such that a user can select which agility action he would like to attempt. This would coexist with the CyFiA's true nature of supplying suggested courses of action to a user to be implemented instead of making decisions strictly based on cost. Implementing this type of program flow would require more beliefs to be made to determine whether an action had been successfully completed. Moreover, a failure-mode agility operation will be added for the situation in which no agility maneuvers are successful and the cyber operation has no chance of success. Finally, the CyFiA will be changed to output agility proposals instead of outputting agility actions. This will cut the CyFiA's ties with the CyAMS's patching simulation.

8. Conclusions

The CyFiA is a decision-support program that works with the cyber-network modeling and simulation system, CyAMS. The CyFiA proposes agility maneuvers to a user based on cost and utility of the maneuver as well as the network and node facts. The control flow of the program models a human OODA loop. The OODA loop has been used as a model to describe decision-making in military environments, making it a reasonable model to use in the CyFiA. The CyFiA was tested to accomplish a patch-management mission while securing a critical path. As a first proof of concept a simulation with a network of 10 nodes and 4 possible agility maneuvers were used. In the future more agility maneuvers will be added and network complexity increased. It is projected that a tool like CyFiA will provide network analysts and cyber teams with a decision aid to evaluate and apply various agility maneuvers while accomplishing cyber missions in a fast-changing, dynamic environment.

9. References

1. Smith D. The pilot's associate program. [accessed 2014 Aug 4]. http://www.dms489.com/PA/PA_index.html, 2004.

2. Buchler N, Marusich L, Sokoloff S. The warfighter associate: decision-support software agent for the management of intelligence, surveillance, and reconnaissance (ISR) assets. Proceedings of SPIE vol. 9079, Ground/Air Multisensor Interoperability, Integration, and Networking for Persistent ISR V; 2014 June 10.

3. Buchler N, Marusich L, Bakdash J, Sokoloff S, Hamm R. The warfighter associate: objective and automated metrics for mission command. [accessed 2014 Aug]. http://www.asinc.com/downloads/WA_Paper.pdf, 2013.

4. Erbs A, Marvel L. Cost computations for cyber fighter associate. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2015 May. Report No.: ARL-TN-0674.

5. Harman D, Brown S, Henz B, Marvel L. A communication protocol for CyAMS and the cyber fighter associate interface. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2015 May. Report No. ARL-TN-0673.

6. Druzdzel M, Flynn R. Decision support systems. In: Encyclopedia of library and information science. 2nd ed. New York (NY): Marcel Dekker, Inc.; 2002.

INTENTIONALLY LEFT BLANK.

Appendix. Code Listing

This appendix appears in its original form, without editorial change.

CyberFighterAssociate.java (main)

```
package com.asinc.cyfia;
import com.asinc.solomon.engine.Engine;
import com.asinc.solomon.engine.IOutputProvider;
import com.asinc.solomon.engine.ThreadedOutputProvider;
import com.cyfia.kb.ExecuteMission;
import java.io.PrintStream;
public class CyberFighterAssociate
{

public static void main(String[] args)
{
Engine theEngine = null;
IOutputProvider outputProvider = new ThreadedOutputProvider("output", 20000);
NotificationHandler = new NotificationHandler();
ProposalHandler = new ProposalHandler();
CyFiAJob jobHandler = new CyFiAJob();
ActionsHandler actionHandler = new ActionsHandler();
outputProvider.setNotificationListener(notificationHandler);
outputProvider.setProposalListener(proposalHandler);
outputProvider.setActionListener(actionHandler);
theEngine = new Engine("CyFiA Test", null, outputProvider);
```

```
jobHandler.setEngine(theEngine);
actionHandler.setEngine(theEngine);
ExecuteMission rootPlan = new ExecuteMission();
rootPlan.setRole("Maintain Capability");
theEngine.updateNode(rootPlan);
Thread jobs = new Thread(jobHandler);
jobs.start();
System.out.println("The Engine has gotten this far");
}

}
```

ActionsHandler.java

```
package com.asinc.cyfia;
import com.asinc.solomon.engine.Action;
import com.asinc.solomon.engine.Engine;
import com.asinc.solomon.engine.QueueListener;
import com.cyfia.kb.BestThroughputCost;
import com.cyfia.kb.CheckCriticalNodes;
import com.cyfia.kb.HealSoftwareCost;
import com.cyfia.kb.IPBlockCost;
import com.cyfia.kb.NodeAttributeUpdate;
import com.cyfia.kb.StartActionSender;
import com.cyfia.kb.UpdateNetworkHealth;
import com.cyfia.kb.WallOffCost;
import java.io.IOException;
```

```
import java.io.PrintStream;
import java.net.DatagramSocket;
import java.net.PortUnreachableException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.BlockingQueue;

public class ActionsHandler implements QueueListener<Action>
{
Engine engine = null;
Database database = Database.getInstance();
RequestHandler requestHandler = RequestHandler.getInstance(this.engine);
String[] parseString;
List<AutomatedAction> actionList = new ArrayList<AutomatedAction>();

public void setEngine(Engine e) {this.engine = e;}
public void process(Action action)
{
Request request = new Request();
if ((action instanceof CheckCriticalNodes))
{
System.out.println("Check Critical Nodes Executing");
CheckCriticalNodes graphCheck = (CheckCriticalNodes)action;
Node affectedNode = this.database.getNode(graphCheck.getNodeId());
List<Node> connectionsList = affectedNode.getConnections();
```

```
Iterator<Node> listIter = connectionsList.iterator();

boolean flag = false;


while (listIter.hasNext())

{

Node criticalNode = (Node)listIter.next();



if ((criticalNode.getNodeCriticality()) &&
(!criticalNode.getNodeHealth().equalsIgnoreCase("immune")))

{

flag = true;

NodeAttributeUpdate nodeToUpdate = new NodeAttributeUpdate();

nodeToUpdate.setNodeId(criticalNode.getNodeId());

nodeToUpdate.setRequestId(graphCheck.getRequestId());

nodeToUpdate.setVulnerabilitySignature(graphCheck.getVulnerbilitySignature());

nodeToUpdate.setAttackType(graphCheck.getAttackType());

nodeToUpdate.setNodeHealth(criticalNode.getNodeHealth());


if (this.requestHandler.queue.offer(nodeToUpdate))

{

System.out.println("Added NodeAttributeUpdate belief to queue for node " +
nodeToUpdate.getNodeId());

}

else

{

System.out.println("Failed to add critical belief to queue");

}
```

```
}

}

if (!flag)
{
LinkedList<Node> nodeList = this.database.getNodeList();
listIter = nodeList.iterator();

while (listIter.hasNext())
{
Node criticalNode = (Node)listIter.next();

if (criticalNode.getNodeCriticality())
{
NodeAttributeUpdate nodeToUpdate = new NodeAttributeUpdate();
nodeToUpdate.setNodeId(criticalNode.getNodeId());
nodeToUpdate.setCapability(String.valueOf(criticalNode.getCapMetric()));
nodeToUpdate.setRequestId(graphCheck.getRequestId());
nodeToUpdate.setAttackType(graphCheck.getAttackType());
nodeToUpdate.setVulnerabilitySignature(graphCheck.getVulnerabilitySignature()
);
nodeToUpdate.setNodeHealth(nodeToUpdate.getNodeHealth());

if (this.requestHandler.queue.offer(nodeToUpdate))
{
System.out.println("Added NodeAttributeUpdate belief to queue for node " +
nodeToUpdate.getNodeId());
```

```
}
else
{
System.out.println("Failed to add critical belief to queue");
}

}

}

}

} // This bracket closes off the CheckCriticalNodes action

else if ((action instanceof UpdateNetworkHealth))
{
System.out.println("Update Network Health Executing");
UpdateNetworkHealth affectedNode = (UpdateNetworkHealth)action;
Node compNode = this.database.getNode(affectedNode.getNodeId());
Iterator<Node> databaseIter = this.database.getNodeList().iterator();
compNode.setNodeHealth(affectedNode.getNodeHealth());

if (affectedNode.getNodeHealth().equalsIgnoreCase("infected"))
{

while (databaseIter.hasNext())
{
```

```
Node nodeToUpdate = (Node)databaseIter.next();


if ((!nodeToUpdate.getNodeHealth().equalsIgnoreCase("immune")) &&
(this.database.isConnected(compNode, nodeToUpdate)) &&
(!nodeToUpdate.getNodeHealth().equalsIgnoreCase("infected")))

{

nodeToUpdate.setNodeHealth("susceptible");

System.out.println("Changed node " + nodeToUpdate.getNodeId() + " health to "
+ nodeToUpdate.getNodeHealth());

}

else if ((!nodeToUpdate.getNodeHealth().equalsIgnoreCase("Immune")) &&
(!nodeToUpdate.getNodeHealth().equalsIgnoreCase("infected")))

{

nodeToUpdate.setNodeHealth("vulnerable");

System.out.println("Changed node " + nodeToUpdate.getNodeId() + " health to "
+ nodeToUpdate.getNodeHealth());

}


}


}


} // This bracket closes off the UpdateNetworkHealth action


else if ((action instanceof IPBlockCost))

{

System.out.println("IP Block Cost Executing");

try
```

```
{
DatagramSocket socket = new DatagramSocket(3081);
DatagramSocket sendSocket = new DatagramSocket(3080);
IPBlockCost ipBlockCost = (IPBlockCost)action;
request.createRequestForSend(ipBlockCost.getRequestId(),
ipBlockCost.getStartNodeId(), ipBlockCost.getEndNodeId(), 1, 0.0D, false);
this.requestHandler.sendRequest(request, sendSocket);
System.out.println("Sent request for IP Block Cost");
this.requestHandler.receiveRequest(socket, this.parseString);
System.out.println("Received cost for IP Block");

if (this.parseString == null)
{
return;
}

AutomatedAction aAction = new AutomatedAction();
aAction.setActionAttributes(this.parseString[0], this.parseString[1],
this.parseString[2], Double.valueOf(this.parseString[3]).doubleValue(), "block");
this.actionList.add(aAction);
}
catch (PortUnreachableException e)
{
e.printStackTrace();
}
catch (IOException e)
{
e.printStackTrace();
```

```
}
catch (Exception e)
{
e.printStackTrace();
}

} // This bracket closes off IPBlockCost action

else if ((action instanceof WallOffCost))
{
System.out.println("Quarantine Function Cost Executing");

try
{
DatagramSocket socket = new DatagramSocket(3081);
DatagramSocket sendSocket = new DatagramSocket(3080);
WallOffCost wallOffCost = (WallOffCost)action;
request.createRequestForSend(wallOffCost.getRequestId(),
wallOffCost.getNodeId(), wallOffCost.getNodeId(), 2, 0.0D, false);
this.requestHandler.sendRequest(request, sendSocket);
System.out.println("Sent request for Wall Off Cost");
this.requestHandler.receiveRequest(socket, this.parseString);
System.out.println("Received cost for Walling off");

if (this.parseString == null)
{
return;
```

```
}


AutomatedAction aAction = new AutomatedAction();

aAction.setActionAttributes(this.parseString[0], this.parseString[1],
this.parseString[2], Double.valueOf(this.parseString[3]).doubleValue(),
"walloff");

this.actionList.add(aAction);

}

catch (PortUnreachableException e)

{

e.printStackTrace();

}

catch (IOException e)

{

e.printStackTrace();

}

catch (Exception e)

{

e.printStackTrace();

}


} // This bracket closes off the WallOffCost action


else if ((action instanceof BestThroughputCost))

{

System.out.println("Patch File Cost Executing");


try
```

```
{
BestThroughputCost bestThroughputCost = (BestThroughputCost)action;
DatagramSocket socket = new DatagramSocket(3081);
DatagramSocket sendSocket = new DatagramSocket(3080);
request.createRequestForSend(bestThroughputCost.getRequestId(),
bestThroughputCost.getEndNodeId(), bestThroughputCost.getStartNodeId(), 5,
Double.valueOf(bestThroughputCost.getPatchSize()).doubleValue(), false);
this.requestHandler.sendRequest(request, sendSocket);
System.out.println("Sent request for Patching Cost");
this.requestHandler.receiveRequest(socket, this.parseString);
System.out.println("Received cost for patching");

if (this.parseString == null)
{
return;
}

AutomatedAction aAction = new AutomatedAction();
aAction.setActionAttributes(this.parseString[0], this.parseString[1],
this.parseString[2], Double.valueOf(this.parseString[3]).doubleValue(), "patch");
this.actionList.add(aAction);
}
catch (PortUnreachableException e)
{
e.printStackTrace();
}
catch (IOException e)
{
```

```
e.printStackTrace();
}
catch (Exception e)
{
e.printStackTrace();
}

} // This bracket closes off the BestThroughputCost action

else if ((action instanceof HealSoftwareCost))
{
System.out.println("Heal Software Cost Executing");

try
{
HealSoftwareCost healSoftwareCost = (HealSoftwareCost)action;
DatagramSocket socket = new DatagramSocket(3081);
DatagramSocket sendSocket = new DatagramSocket(3080);
request.createRequestForSend(healSoftwareCost.getRequestId(),
healSoftwareCost.getNodeId(), healSoftwareCost.getNodeId(), 3, 0.0D, false);
this.requestHandler.sendRequest(request, sendSocket);
System.out.println("Sent request for Heal Software Cost");
this.requestHandler.receiveRequest(socket, this.parseString);
System.out.println("Received cost to Heal Software");

if (this.parseString == null)
{
```

```
return;
}

AutomatedAction aAction = new AutomatedAction();
aAction.setActionAttributes(this.parseString[0], this.parseString[1],
this.parseString[2], Double.valueOf(this.parseString[3]).doubleValue(), "heal");
this.actionList.add(aAction);
}
catch (PortUnreachableException e)
{
e.printStackTrace();
}
catch (IOException e)
{
e.printStackTrace();
}
catch (Exception e)
{
e.printStackTrace();
}

} // This bracket closes off the HealSoftwareCost action

else if ((action instanceof StartActionSender))
{
System.out.println("Action Sender Start Execution");
StartActionSender startActionSender = (StartActionSender)action;
```

```
AutomatedActionSender automatedActionSender =
AutomatedActionSender.getInstance(this.engine);

automatedActionSender.setRequestId(startActionSender.getRequestId());

automatedActionSender.getNumActions();

}


} // This bracket closes off the StartActionSender action


}
```

AutomatedAction.java

```
package com.asinc.cyfia;


public class AutomatedAction

{

private String requestId;

private String startNodeId;

private String endNodeId;

private String actionName;

private double cost;


public void setActionAttributes(String requestId, String startNodeId, String
endNodeId, double cost, String actionName)

{

this.requestId = requestId;

this.startNodeId = startNodeId;

this.endNodeId = endNodeId;
```

```
this.cost = cost;
this.actionName = actionName;
}

public String getRequestId()
{
return this.requestId;
}

public String getStartNodeId()
{
return this.startNodeId;
}

public String getEndNodeId()
{
return this.endNodeId;
}

public String getActionName()
{
return this.actionName;
}

public double getCost()
{
return this.cost;
```

```
}


public String toString()

{

String str = this.requestId + "," + this.startNodeId + "," + this.actionName + "," +
this.endNodeId;

return str;

}


}
```

AutomatedActionSender.java

```
package com.asinc.cyfia;

import com.asinc.solomon.engine.Engine;
import com.cyfia.kb.IPBlockSuccess;
import com.cyfia.kb.IPblocked;
import com.cyfia.kb.SoftwareActionSuccess;
import com.cyfia.kb.SoftwareSecured;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.BlockingQueue;

public class AutomatedActionSender
{
static AutomatedActionSender instance = null;
private static List<AutomatedAction> actionList = new
ArrayList<AutomatedAction>();
private static int numActions;
private String requestId;
private static Engine engine = null;
static RequestHandler requestHandler = RequestHandler.getInstance(engine);

public static AutomatedActionSender getInstance(Engine e)
```

```
{
if (instance == null)
{
engine = e;
instance = new AutomatedActionSender();
}
return instance;
}

public void getNumActions()
{
SoftwareSecured query = new SoftwareSecured();
IPblocked query1 = new IPblocked();
query1.setRequestId(this.requestId);
query.setRequestId(this.requestId);
List<SoftwareSecured> resultList = engine.find(query);
List<IPblocked> resultList1 = engine.find(query1);

if ((resultList.size() == 1) && (resultList1.size() == 1))
{

if (((SoftwareSecured)resultList.get(0)).getPatchFile() != null) {
numActions = 4;
}
else
{
numActions = 3;
}

}
else
{
numActions = 2;
}

actionList.clear();
}

public void setRequestId(String requestId)
{
this.requestId = requestId;
}

public static void sendAction(List<AutomatedAction> actionList)
{
```

```
for (int i = 0; i < numActions; i++)
{
actionList.add((AutomatedAction)actionList.get(i));
}

if (actionList.size() == numActions)
{
sortList(actionList);

try
{
DatagramSocket socket = new DatagramSocket(3012);
byte[] sendArray = new byte[1024];
String str = ((AutomatedAction)actionList.get(0)).toString();
System.arraycopy(str.getBytes(), 0, sendArray, 0, str.length());
InetAddress outgoingThreadAddress = InetAddress.getLocalHost();
DatagramPacket sendPacket = new DatagramPacket(sendArray,
sendArray.length, outgoingThreadAddress, 3011);

while (!socket.isClosed())
{
socket.send(sendPacket);
socket.close();
}

}
catch (SocketException e)
{
e.printStackTrace();
}
catch (UnknownHostException e)
{
e.printStackTrace();
}
catch (IOException e)
{
e.printStackTrace();
}
if
(((AutomatedAction)actionList.get(0)).getActionName().equalsIgnoreCase("bloc
k"))
{
IPBlockSuccess updateNode = new IPBlockSuccess();
AutomatedAction action = (AutomatedAction)actionList.get(0);
updateNode.setRequestId(action.getRequestId());
updateNode.setInfNode(action.getStartNodeId());
updateNode.setVulnNode(action.getEndNodeId());
```

```
requestHandler.queue.add(updateNode);
}
else
{
SoftwareActionSuccess updateNode = new SoftwareActionSuccess();
AutomatedAction action = (AutomatedAction)actionList.get(0);
updateNode.setRequestId(action.getRequestId());
updateNode.setNewImmNode(action.getEndNodeId());
requestHandler.queue.add(updateNode);
}

}
}
public static void sortList(List<AutomatedAction> list)
{

if (((AutomatedAction)list.get(0)).getCost() >
((AutomatedAction)list.get(1)).getCost())
{
AutomatedAction tempAction = (AutomatedAction)list.get(0);
list.set(0, (AutomatedAction)list.get(1));
list.set(1, tempAction);
}

if (((AutomatedAction)list.get(0)).getCost() >
((AutomatedAction)list.get(2)).getCost())
{
AutomatedAction tempAction = (AutomatedAction)list.get(0);
list.set(0, (AutomatedAction)list.get(2));
list.set(2, tempAction);
}

if (((AutomatedAction)list.get(0)).getCost() >
((AutomatedAction)list.get(3)).getCost())
{
AutomatedAction tempAction = (AutomatedAction)list.get(0);
list.set(0, (AutomatedAction)list.get(3));
list.set(3, tempAction);
}

}

}
```

CyFiAJob.java

```
package com.asinc.cyfia;
```

```
import com.asinc.solomon.engine.Engine;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.BlockingQueue;

public class CyFiAJob implements Runnable
{
private RequestHandler requestHandler = null;
private Engine engine = null;
List<Thread> threadList = new ArrayList<Thread>();

public CyFiAJob()
{

if (this.requestHandler == null)
{
this.requestHandler = RequestHandler.getInstance(this.engine);
}

}

public RequestHandler getRequestHandler()
{
return this.requestHandler;
}

public void setEngine(Engine e)
{
this.engine = e;
}

public void run()
{
boolean beginRun = true;

Thread sL5 = new Thread(new SocketListener(3041, this.engine));
this.threadList.add(sL5);
Thread sL6 = new Thread(new SocketListener(3081, this.engine));
this.threadList.add(sL6);
Thread sL1 = new Thread(new SocketListener(3061, this.engine));
this.threadList.add(sL1);
Thread sL2 = new Thread(new SocketListener(3051, this.engine));
this.threadList.add(sL2);
Thread sL3 = new Thread(new SocketListener(3071, this.engine));
this.threadList.add(sL3);
Thread sL4 = new Thread(new SocketListener(3031, this.engine));
```

```
this.threadList.add(sL4);

for (Thread t : this.threadList)
{
t.start();
}

while ((beginRun) || (this.requestHandler.queue.size() > 0))
{

if (this.requestHandler.queue.size() > 0)
{
this.requestHandler.makeBelief(this.engine);
}

}

}

}
```

Database.java

```
package com.asinc.cyfia;

//import java.io.PrintStream;
import java.util.Iterator;
import java.util.LinkedList;
//import java.util.List;

public class Database
{
public LinkedList<String> criticalPath = new LinkedList<String>();
private LinkedList<Node> nodeList = new LinkedList<Node>();
int numNodes = -1;
static Database instance = null;

public static Database getInstance()
{

if (instance == null)
{
instance = new Database();
}

return instance;
```

```
}

public void changeCriticalPath(LinkedList<String> list)
{
this.criticalPath.clear();
this.criticalPath.addAll(list);
}

public void changeCriticalPath(String node, int index)
{
this.criticalPath.set(index, node);
}

public void changeCriticalPath(LinkedList<String> list, int startIndex, int
endIndex)
{

if ((startIndex >= 0) && (endIndex >= 0) && (startIndex != endIndex) &&
(endIndex > startIndex))
{

if ((startIndex < this.criticalPath.size()) && (endIndex < this.criticalPath.size()))
{

for (int i = startIndex; i < endIndex; i++)
{
this.criticalPath.set(i, (String)list.get(i));
}

}
else
{
System.out.println("Index out of bounds. Did not create new critical path.");
}

}
else
{
System.out.println("Invalid indices or start and end index are the same.");
}

}

public void printCriticalPath()
{
Iterator<String> iter = this.criticalPath.iterator();
while (iter.hasNext())
```

```
{
System.out.println((String)iter.next());
}

}

public void printNodeList()
{
Iterator<Node> iter = this.nodeList.iterator();

while (iter.hasNext())
{
System.out.println(((Node)iter.next()).getNodeId());
}

}

public void setTotalNodes(int numNodes)
{

if (numNodes > 0)
{
this.numNodes = numNodes;
}
else
{
System.out.println("Cannot make network of 0 or negative nodes");
}

}

public void addNode(Node node)
{
this.nodeList.add(node);
}

public void updateNodeCapability(String nodeId, int capability, String os)
{
Iterator<Node> listIterator = this.nodeList.iterator();
boolean flag = true;
Node nodeToUpdate = null;

while ((listIterator.hasNext()) && (flag))
{
nodeToUpdate = (Node)listIterator.next();
```

```
if (nodeToUpdate.getNodeId() == nodeId)
{
flag = false;

if (capability >= 0)
{
nodeToUpdate.setCapabilityMetric(capability);
}

if (os != null)
{
nodeToUpdate.setOperatingSystem(os);
}

}

}

}

public void updateNodeHealth(String nodeId, String newHealth)
{
Iterator<Node> listIterator = this.nodeList.iterator();
boolean flag = true;
Node nodeToUpdate = null;

while ((listIterator.hasNext()) && (flag))
{
nodeToUpdate = (Node)listIterator.next();

if (nodeToUpdate.getNodeId() == nodeId)
{
nodeToUpdate.setNodeHealth(newHealth);
flag = false;
}

}

}

public void updateNodeCriticality(String nodeId, boolean newCriticality)
{
Iterator<Node> listIterator = this.nodeList.iterator();
boolean flag = true;
Node nodeToUpdate = null;
```

```
while ((listIterator.hasNext()) && (flag))
{
nodeToUpdate = (Node)listIterator.next();

if (nodeToUpdate.getNodeId() == nodeId)
{
nodeToUpdate.setNodeCriticality(newCriticality);
flag = false;
}

}

}

public void updateGeo(String nodeId, double newLat, double newLon)
{
Iterator<Node> listIterator = this.nodeList.iterator();
boolean flag = true;
Node nodeToUpdate = null;

while ((listIterator.hasNext()) && (flag))
{
nodeToUpdate = (Node)listIterator.next();

if (nodeToUpdate.getNodeId() == nodeId)
{
nodeToUpdate.setGeo(newLat, newLon);
flag = false;
}

}

}

public void updateBattery(String nodeId, int total, double power, double transfer,
double receive)
{
Iterator<Node> listIterator = this.nodeList.iterator();
boolean flag = true;
Node nodeToUpdate = null;

while ((listIterator.hasNext()) && (flag))
{
nodeToUpdate = (Node)listIterator.next();

if (nodeToUpdate.getNodeId() == nodeId)
{
```

```
if (total >= 0)
{
nodeToUpdate.setBatteryTotal(total);
}

if (power >= 0.0D)
{
nodeToUpdate.setBatteryPower(power);
}

if (transfer >= 0.0D)
{
nodeToUpdate.setBatteryTransfer(transfer);
}

if (receive >= 0.0D)
{
nodeToUpdate.setBatteryReceive(receive);
}

flag = false;
}

}

}

public Node getNode(String nodeID)
{
Iterator<Node> listIterator = this.nodeList.iterator();
boolean flag = true;
Node nodeToGet = null;

while ((listIterator.hasNext()) && (flag))
{
Node checkNode = (Node)listIterator.next();

if (checkNode.getNodeId().equals(nodeID))
{
nodeToGet = checkNode;
flag = false;
}

}
return nodeToGet;
```

```
}

public LinkedList<Node> getNodeList()
{
return this.nodeList;
}

public boolean isConnected(Node a, Node b)
{

if (a.getConnections().contains(b))
{
return true;
}

return false;
}

public boolean isCritical(Node a)
{
return a.getNodeCriticality();
}

public boolean checkImmunity()
{
Iterator<String> listIter = this.criticalPath.iterator();
Iterator<Node> nodeIter = this.nodeList.iterator();
boolean flag = true;

while (listIter.hasNext() && nodeIter.hasNext() && (flag))
{
Node checkNode = (Node)nodeIter.next();

if ((checkNode.getNodeId() == listIter.next()) &&
(!(checkNode.getNodeHealth().equalsIgnoreCase("immune"))))
{
flag = false;
}

}

return flag;
}

public void addConnection(Node a, String nodeId)
{
```

```
a.setConnections(getNode(nodeId));
}

public LinkedList<String> getInfected(String nodeId)
{
LinkedList<String> resultList = new LinkedList<String>();
Iterator<Node> listIter = this.nodeList.iterator();

while (listIter.hasNext())
{
Node infNode = (Node)listIter.next();

if ((infNode.getNodeHealth().equalsIgnoreCase("infected")) &&
(!infNode.getNodeId().equalsIgnoreCase(nodeId)))
{
resultList.add(infNode.getNodeId());
}

}

return resultList;
}

public LinkedList<String> getImmune()
{
LinkedList<String> resultList = new LinkedList<String>();
Iterator<Node> listIter = this.nodeList.iterator();

while (listIter.hasNext())
{
Node immNode = (Node)listIter.next();

if (immNode.getNodeHealth().equalsIgnoreCase("immune")) {}
resultList.add(immNode.getNodeId());

}

return resultList;
}

public boolean criticalPathPatched()
{
Iterator<String> critPathIt = this.criticalPath.iterator();
boolean flag = true;

while ((critPathIt.hasNext()) && (flag))
```

```
{
String str = (String)critPathIt.next();
Node checkNode = getNode(str);

if (!checkNode.getNodeHealth().equalsIgnoreCase("immune"))
{
flag = false;
}

}

return flag;
}

public void removeConnections(String infNode, String vulnNode)
{
Node node = getNode(infNode);
Node node1 = getNode(vulnNode);
node.removeConnection(node1);
node1.removeConnection(node);
}

}
```

Node.java

```
package com.asinc.cyfia;
import java.util.ArrayList;
import java.util.List;

public class Node
{
private String nodeId;
private String operatingSystem;
private String nodeHealth;
private int capabilityMetric;
private double[] geo = new double[2];
private int batteryTotal = 100;
private double batteryPower = 100.0D;
private double computationRate;
private double batteryTransferRate;
private double batteryReceiveRate;
private List<Node> connections = new ArrayList<Node>();
private boolean originNode = false;
private boolean nodeCriticality = false;
```

```
public void setNodeId(String nodeId)
{
this.nodeId = nodeId;
}

public void setOperatingSystem(String os)
{
this.operatingSystem = os;
}

public void setNodeHealth(String nodeHealth)
{
this.nodeHealth = nodeHealth;
}

public void setNodeCriticality(boolean criticality)
{
this.nodeCriticality = criticality;
}

public void setCapabilityMetric(int cap)
{
this.capabilityMetric = cap;
}

public void setGeo(double lat, double lon)
{
this.geo[0] = lat;
this.geo[1] = lon;
}

public void setBatteryTotal(int batTotal)
{
this.batteryTotal = batTotal;
}

public void setBatteryPower(double batPower)
{
this.batteryPower = batPower;
}

public void setCompRate(double compRate)
{
this.computationRate = compRate;
}
```

```
public void setBatteryTransfer(double btr)
{
this.batteryTransferRate = btr;
}

public void setBatteryReceive(double brr)
{
this.batteryReceiveRate = brr;
}

public void setConnections(Node a)
{

if ((a != null) && (this.connections != null))
{
this.connections.add(a);
}

}

public void setOriginNode()
{
this.originNode = true;
}

public void removeConnection(Node a)
{
this.connections.remove(a);
}

public String getNodeId()
{
return this.nodeId;
}

public String getOperatingSystem()
{
return this.operatingSystem;
}

public String getNodeHealth()
{
return this.nodeHealth;
}

public boolean getNodeCriticality()
```

```
{
return this.nodeCriticality;
}

public int getCapMetric()
{
return this.capabilityMetric;
}

public int getBatteryTotal()
{
return this.batteryTotal;
}

public double getBatteryPower()
{
return this.batteryPower;
}

public String getGeo()
{
String str = new String("<" + this.geo[0] + ", " + this.geo[1] + ">");
return str;
}

public double getCompRate()
{
return this.computationRate;
}

public double getBatTrans()
{
return this.batteryTransferRate;
}

public double getBatRec()
{
return this.batteryReceiveRate;
}

public List<Node> getConnections()
{
return this.connections;
}

public boolean getOriginNode()
```

```
{
return this.originNode;
}

public boolean isConnected(Node a)
{

if ((a == null) || (this.connections == null))
{
return false;
}

return this.connections.contains(a);
}

}
```

NotificationHandler.java

```
package com.asinc.cyfia;

import com.asinc.solomon.engine.Notification;
import com.asinc.solomon.engine.QueueListener;
import com.cyfia.kb.nodeIsInfected;
import com.cyfia.kb.nodeIsSusceptible;
import com.cyfia.kb.nodeIsVulnerable;
import com.cyfia.kb.requestCreated;
import java.io.PrintStream;

public class NotificationHandler implements QueueListener<Notification>
{

public void process(Notification notification)
{
System.out.println("The Notification Handler has started");
System.out.println("The thread is: " + Thread.currentThread());

if ((notification instanceof nodeIsVulnerable))
{
nodeIsVulnerable myNotification = (nodeIsVulnerable)notification;
System.out.println("\n" + myNotification.getNotificationDescription());
System.out.println("Node ID: " + myNotification.getNodeId());
System.out.println("Node's health has changed to:" +
myNotification.getNodeHealth());
}
else if ((notification instanceof nodeIsInfected))
{
```

```
nodeIsInfected myNotification = (nodeIsInfected)notification;
System.out.println("\n" + myNotification.getNotificationDescription());
System.out.println("Node ID: " + myNotification.getNodeId());
System.out.println("Node's health has changed to: " +
myNotification.getNodeHealth());
}
else if ((notification instanceof nodeIsSusceptible))
{
nodeIsSusceptible myNotification = (nodeIsSusceptible)notification;
System.out.println("\n" + myNotification.getNotificationDescription());
System.out.println("Node ID: " + myNotification.getNodeId());
System.out.println("Node's health has changed to: " +
myNotification.getNodeHealth());
}
else if ((notification instanceof requestCreated))
{
requestCreated myNotification = (requestCreated)notification;
System.out.println(myNotification.getNotificationDescription());
}

}

}
```

PatchFile.java

```
package com.asinc.cyfia;

import java.util.Random;

public class PatchFile
{
protected String patchFile;
protected float patchSize;
protected String originNode;
Random rand = new Random();

public PatchFile()
{
this.patchFile = "Chicken Noodle Soup.forthecomputersoul";
this.patchSize = this.rand.nextFloat();
}

public String getPatchFile()
{
return this.patchFile;
}
```

```
public float getPatchSize()
{
return this.patchSize;
}

}
```

ProposalHandler.java

```
package com.asinc.cyfia;

import com.asinc.solomon.engine.Plan;
import com.asinc.solomon.engine.QueueListener;
import com.cyfia.kb.BlockIP;
import com.cyfia.kb.HealSoftware;
import com.cyfia.kb.PatchFile;

public class ProposalHandler implements QueueListener<Plan>
{
public void process(Plan plan)
{

//if (!(plan instanceof PatchFile))
{

//if (!(plan instanceof BlockIP))
{
// (plan instanceof HealSoftware);
}

}
// I'm not sure if I used this for anything. I think at some point I decided proposals
were not needed because the simulation was completely autonomous
```

Request.java

```
package com.asinc.cyfia;

import java.io.PrintStream;

public class Request
{
private String requestId = null;
private String startNodeId = null;
private String endNodeId = null;
private double patchSize = 0.0D;
```

```
private int planNum;
private boolean isBat = false;

public void createRequestForSend(String requestId, String startNodeId, String
endNodeId, int planNum, double patchSize, boolean battery)
{

if ((planNum >= 1) && (planNum <= 5))
{
this.requestId = requestId;
this.startNodeId = startNodeId;
this.endNodeId = endNodeId;
this.patchSize = patchSize;
this.planNum = planNum;
this.isBat = battery;
}
else
{
System.out.println("Plan index out of bounds");
}

}

public String getRequestId()
{
return this.requestId;
}

public String getStartNodeId()
{
return this.startNodeId;
}

public String getEndNodeId()
{
return this.endNodeId;
}

public double getPatchSize()
{
return this.patchSize;
}

public int getPlanNum()
{
return this.planNum;
```

```
}

public Request getRequest()
{
return this;
}

public boolean getIsBat()
{
return this.isBat;
}

public String toString()
{
String str = this.requestId + "," + this.startNodeId + "," + this.endNodeId + "," +
this.planNum + "," + this.patchSize + "," + this.isBat;
return str;
}

}
```

RequestHandler.java

```
package com.asinc.cyfia;

import com.asinc.solomon.engine.Engine;
import com.asinc.solomon.engine.Node;
import com.cyfia.kb.IPBlockSuccess;
import com.cyfia.kb.NodeAttributeUpdate;
import com.cyfia.kb.NodeBatteryUpdate;
import com.cyfia.kb.NodeCapabilitiesUpdate;
import com.cyfia.kb.NodeGeoUpdate;
import com.cyfia.kb.NodeHealthChange;
import com.cyfia.kb.SoftwareActionSuccess;
import java.io.IOException;
import java.io.PrintStream;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.PortUnreachableException;
import java.net.SocketTimeoutException;
import java.util.LinkedList;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class RequestHandler
```

```
{
BlockingQueue<Request> requestQueue = new
LinkedBlockingQueue<Request>();
BlockingQueue<Node> queue = new LinkedBlockingQueue<Node>();
static RequestHandler instance = null;
Database database = Database.getInstance();
static Engine engine = null;

public static RequestHandler getInstance(Engine e)
{

if (instance == null)
{
engine = e;
instance = new RequestHandler();
}

return instance;
}

public void makeBelief(Engine engine)
{

try
{

if ((this.queue.peek() instanceof NodeAttributeUpdate))
{
NodeAttributeUpdate belief = (NodeAttributeUpdate)this.queue.take();
engine.updateNode(belief);
}
else if ((this.queue.peek() instanceof NodeCapabilitiesUpdate))
{
NodeCapabilitiesUpdate belief = (NodeCapabilitiesUpdate)this.queue.take();
engine.updateNode(belief);
}
else if ((this.queue.peek() instanceof NodeHealthChange))
{
NodeHealthChange belief = (NodeHealthChange)this.queue.take();
engine.updateNode(belief);
}
else if ((this.queue.peek() instanceof IPBlockSuccess))
{
IPBlockSuccess belief = (IPBlockSuccess)this.queue.take();
engine.updateNode(belief);
}
else if ((this.queue.peek() instanceof SoftwareActionSuccess))
```

```
{
SoftwareActionSuccess belief = (SoftwareActionSuccess)this.queue.take();
engine.updateNode(belief);
}
else if ((this.queue.peek() instanceof NodeGeoUpdate))
{
NodeGeoUpdate belief = (NodeGeoUpdate)this.queue.take();
engine.updateNode(belief);
}
else if ((this.queue.peek() instanceof NodeBatteryUpdate))
{
NodeBatteryUpdate belief = (NodeBatteryUpdate)this.queue.take();
engine.updateNode(belief);
}

}
catch (InterruptedException e)
{
e.printStackTrace();
}

}

public void sendRequest(Request r, DatagramSocket outputSocket)
{

try
{
outputSocket.setSoTimeout(15000);

String str = r.toString();
byte[] sendArray = new byte[1024];
System.arraycopy(str.getBytes(), 0, sendArray, 0, str.length());
InetAddress outgoingThreadAddress = InetAddress.getLocalHost();
DatagramPacket sendPacket = new DatagramPacket(sendArray,
sendArray.length, outgoingThreadAddress, 3082);

while (!outputSocket.isClosed())
{
outputSocket.send(sendPacket);
System.out.println("Sent string: " + str);
outputSocket.close();
}

}
catch (SocketTimeoutException e)
```

```
{
e.printStackTrace();
}
catch (PortUnreachableException e)
{
e.printStackTrace();
}
catch (IOException e)
{
e.printStackTrace();
}
catch (Exception e)
{
e.printStackTrace();
}

}

public void receiveRequest(DatagramSocket inputThreadSocket,
LinkedList<String> list, String endNode, String[] parseString)
{

try
{
System.out.println("Request listener listening on port 3081");
inputThreadSocket.setSoTimeout(15000);
byte[] receivedData = new byte[1024];
DatagramPacket receivedPacket = new DatagramPacket(receivedData,
receivedData.length);

while (!inputThreadSocket.isClosed())
{
inputThreadSocket.receive(receivedPacket);
String rDataString = receivedData.toString();
parseString = rDataString.split(",");
System.out.println(parseString);

if (!list.contains(parseString[1]))
{
list.add(parseString[1]);
}

if (!list.contains(parseString[2]))
{
list.add(parseString[2]);
}
```

```
if (list.contains(endNode))
{
inputThreadSocket.close();
}

}

}
catch (SocketTimeoutException e)
{
e.printStackTrace();
}
catch (PortUnreachableException e)
{
e.printStackTrace();
}
catch (IOException e)
{
e.printStackTrace();
}
catch (Exception e)
{
e.printStackTrace();
}

}

public void receiveRequest(DatagramSocket inputThreadSocket, String[]
parseString)
{

try
{
System.out.println("Request listener listening on port 3081");
inputThreadSocket.setSoTimeout(15000);
byte[] receivedData = new byte[1024];
DatagramPacket receivedPacket = new DatagramPacket(receivedData,
receivedData.length);

while (!inputThreadSocket.isClosed())
{
inputThreadSocket.receive(receivedPacket);
String rDataString = receivedData.toString();
parseString = rDataString.split(",");
inputThreadSocket.close();
}

}
```

```
catch (SocketTimeoutException e)
{
e.printStackTrace();
}
catch (PortUnreachableException e)
{
e.printStackTrace();
}
catch (IOException e)
{
e.printStackTrace();
}
catch (Exception e)
{
e.printStackTrace();
}

}

}
```

SocketListener.java

```
package com.asinc.cyfia;

import com.asinc.solomon.engine.Engine;
import com.cyfia.kb.NodeBatteryUpdate;
import com.cyfia.kb.NodeCapabilitiesUpdate;
import com.cyfia.kb.NodeGeoUpdate;
import com.cyfia.kb.NodeHealthChange;
import java.io.IOException;
import java.io.PrintStream;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.PortUnreachableException;
import java.net.SocketTimeoutException;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.BlockingQueue;

public class SocketListener implements Runnable
{
private int port;
static List<String> criticalNodeList = new ArrayList<String>();
Engine engine;
```

```
private String inData = null;
private String delims = ",";
private String[] tokens = null;
private static final int MAXREADSIZE = 2048;
int count = 0;
RequestHandler queue = RequestHandler.getInstance(this.engine);
Database database = Database.getInstance();

public SocketListener(int port, Engine e)
{
this.port = port;
this.engine = e;
}

public void setEngine(Engine e)
{
this.engine = e;
}

public void run()
{
boolean run = true;

try
{
System.out.println("The listener has started. Trying to listen on port: " +
this.port);
DatagramSocket listenSocket = new DatagramSocket(this.port);
listenSocket.setReceiveBufferSize(1000000000);
listenSocket.setSoTimeout(120000);
System.out.println("Successfully started listening on port: " + this.port);
byte[] receivedData = new byte[2048];
DatagramPacket receivedPacket = new DatagramPacket(receivedData,
receivedData.length);

while (run)
{
listenSocket.receive(receivedPacket);
System.out.println("Received data on port: " + this.port + "!");

if ((this.port == 3061) && (receivedData.length != 0))
{
setInData(receivedPacket);
NodeCapabilitiesUpdate nodeToUpdate = new NodeCapabilitiesUpdate();
nodeToUpdate.setRequestId(this.tokens[0]);
nodeToUpdate.setNodeId(this.tokens[1]);
nodeToUpdate.setCapability(this.tokens[2]);
nodeToUpdate.setOs(this.tokens[3]);
```

```
if (nodeToUpdate.getRequestId() != null)
{

if (this.queue.queue.offer(nodeToUpdate))
{
System.out.println("Added a belief to the queue.");
}
else
{
System.out.println("Failed to add belief to queue");
}

}
else
{
System.out.println("Failed to make belief: Node Capability Update");
}

}
else if ((this.port == 3051) && (receivedData.length != 0))
{
setInData(receivedPacket);
NodeHealthChange nodeHealthChange = new NodeHealthChange();
nodeHealthChange.setRequestId(this.tokens[0]);
nodeHealthChange.setNodeId(this.tokens[1]);
nodeHealthChange.setNodeHealth(this.tokens[2]);
nodeHealthChange.setAttackType(this.tokens[3]);
nodeHealthChange.setVulnerabilitySignature(this.tokens[4]);
if ((nodeHealthChange.getRequestId() != null) &&
(nodeHealthChange.getNodeHealth() != null) &&
(nodeHealthChange.getAttackType() != null))
{

if (this.queue.queue.offer(nodeHealthChange))
{
System.out.println("Added a belief to the queue");
}
else
{
System.out.println("Failed to add belief to queue");
}

}
else
{
System.out.println("Failed to make belief: Node Health Change");
```

```
}

}
else if ((this.port == 3071) && (receivedData.length != 0))
{
setInData(receivedPacket);
NodeGeoUpdate nodeToUpdate = new NodeGeoUpdate();
nodeToUpdate.setRequestId(this.tokens[0]);
nodeToUpdate.setNodeId(this.tokens[1]);
nodeToUpdate.setLat(Double.valueOf(this.tokens[2]));
nodeToUpdate.setLon(Double.valueOf(this.tokens[3]));

if ((nodeToUpdate.getNodeId() != null) &&
(nodeToUpdate.getLat().doubleValue() >= 0.0D) &&
(nodeToUpdate.getLon().doubleValue() >= 0.0D))
{

if (this.queue.queue.offer(nodeToUpdate))
{
System.out.println("Added belief to the queue");
}
else
{
System.out.println("Failed to add belief to queue");
}

}
else
{
System.out.println("Failed to make belief: Node Geo Update");
}

}
else if ((this.port == 3031) && (receivedData.length != 0))
{
setInData(receivedPacket);
NodeBatteryUpdate nodeToUpdate = new NodeBatteryUpdate();
nodeToUpdate.setRequestId(this.tokens[0]);
nodeToUpdate.setNodeId(this.tokens[1]);
nodeToUpdate.setBatteryLevel(this.tokens[2]);
nodeToUpdate.setBatteryPower(this.tokens[3]);
nodeToUpdate.setCompRate(this.tokens[4]);
nodeToUpdate.setTransferRate(this.tokens[5]);
nodeToUpdate.setReceiveRate(this.tokens[6]);

if ((nodeToUpdate.getBatteryLevel() != null) && (nodeToUpdate.getNodeId() !=
null) && (nodeToUpdate.getRequestId() != null))
```

```
{

if (this.queue.queue.offer(nodeToUpdate))
{
System.out.println("Added belief to queue");
}
else
{
System.out.println("Failed to add belief to queue");
}

}
else
{
System.out.println("Failed to make belief: Node Battery Update");
}

}
else if ((this.port == 3041) && (receivedData.length != 0))
{
setInData(receivedPacket);
this.count += 1;
Node node = this.database.getNode(this.tokens[1]);
Node node1 = this.database.getNode(this.tokens[2]);

if (node == null)
{
node = new Node();
node.setNodeId(this.tokens[1]);
node.setNodeHealth("normal");
this.database.addNode(node);

if (criticalNodeList.contains(this.tokens[1]))
{
this.database.criticalPath.add(this.tokens[1]);
node.setNodeCriticality(true);
}

}

if (node1 == null)
{
node1 = new Node();
node1.setNodeId(this.tokens[2]);
node1.setNodeHealth("normal");
this.database.addNode(node1);
```

```
if (criticalNodeList.contains(this.tokens[2]))
{
this.database.criticalPath.add(this.tokens[2]);
node.setNodeCriticality(true);
}

}

if ((node != null) && (node1 != null) && (!node.isConnected(node1)))
{
this.database.addConnection(node, node1.getNodeId());
}

if (this.count == 22)
{
this.database.printNodeList();
this.database.printCriticalPath();
run = false;
}

}
else if ((this.port == 3081) && (receivedData.length != 0))
{
setInData(receivedPacket);

if (this.database.getNode(this.tokens[1]) != null)
{
this.database.criticalPath.add(this.tokens[1]);
this.database.getNode(this.tokens[1]).setNodeCriticality(true);
}
else
{
criticalNodeList.add(this.tokens[1]);
}

if (this.tokens[2].equals("-1"))
{
listenSocket.close();
this.database.printCriticalPath();
run = false;
}

}

}
```

```
listenSocket.close();
}
catch (SocketTimeoutException e)
{
e.printStackTrace();
}
catch (PortUnreachableException e)
{
e.printStackTrace();
}
catch (IOException e)
{
e.printStackTrace();
}
catch (Exception e)
{
e.printStackTrace();
}

}

public String getInData()
{
return this.inData;
}

public void setInData(DatagramPacket r)
{
this.inData = new String(r.getData());
System.out.println("Received: " + this.inData + " " + this.inData.length());
this.tokens = this.inData.split(this.delims);
}

public int getTokensSize()
{
return this.tokens.length;
}

public String getNewRequestID()
{
String str = Integer.toString(this.count);
this.count += 1;
return str;
}

}
```

INTENTIONALLY LEFT BLANK.

List of Symbols, Abbreviations, and Acronyms

CRA	Collaborative Research Alliance
CSec	Cyber-Security
CyAMS	Cyber Army Modeling and Simulation
CyFiA	Cyber Fighter Associate
D–A	Decide–Act
GUI	graphical user interface
ID	identification
IP	Intrusion Prevention
O–O	Observe–Orient
OODA	Observe, Orient, Decide, Act
PASS	Publish and Subscribe Service
SA	situational awareness
SIIS	Systems and Internet Infrastructure Security
UDP	User Datagram Protocol

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
2 (PDF)	DIR USARL RDRL CIN T A SWAMI L MARVEL